University of Tripoli

Faculty of Engineering

Electrical and Electronic Engineering

Course Number: EE434	**Course Name: Computer Architecture**
Credit Hours: 3	**Prerequisite:** EE334
Lecture time: Monday (9:30-10:45)	**Classroom**: Room 4
Thursday (9:30-10:45)	**Classroom**: Room 4

Text Book:

Computer Organization and Design
The Hardware/Software Interface, 5th Edition
David Patterson & John Hennessy

References:

Computer Architecture, A Quantitative Approach, 3th Edition

John Hennessy & David Patterson

Computer Organization and Architecture, 10th Edition
William Stallings

Computer Architecture

From Microprocessors to Supercomputers

Behrooz Parhami

Computer resources:

MIPS simulators

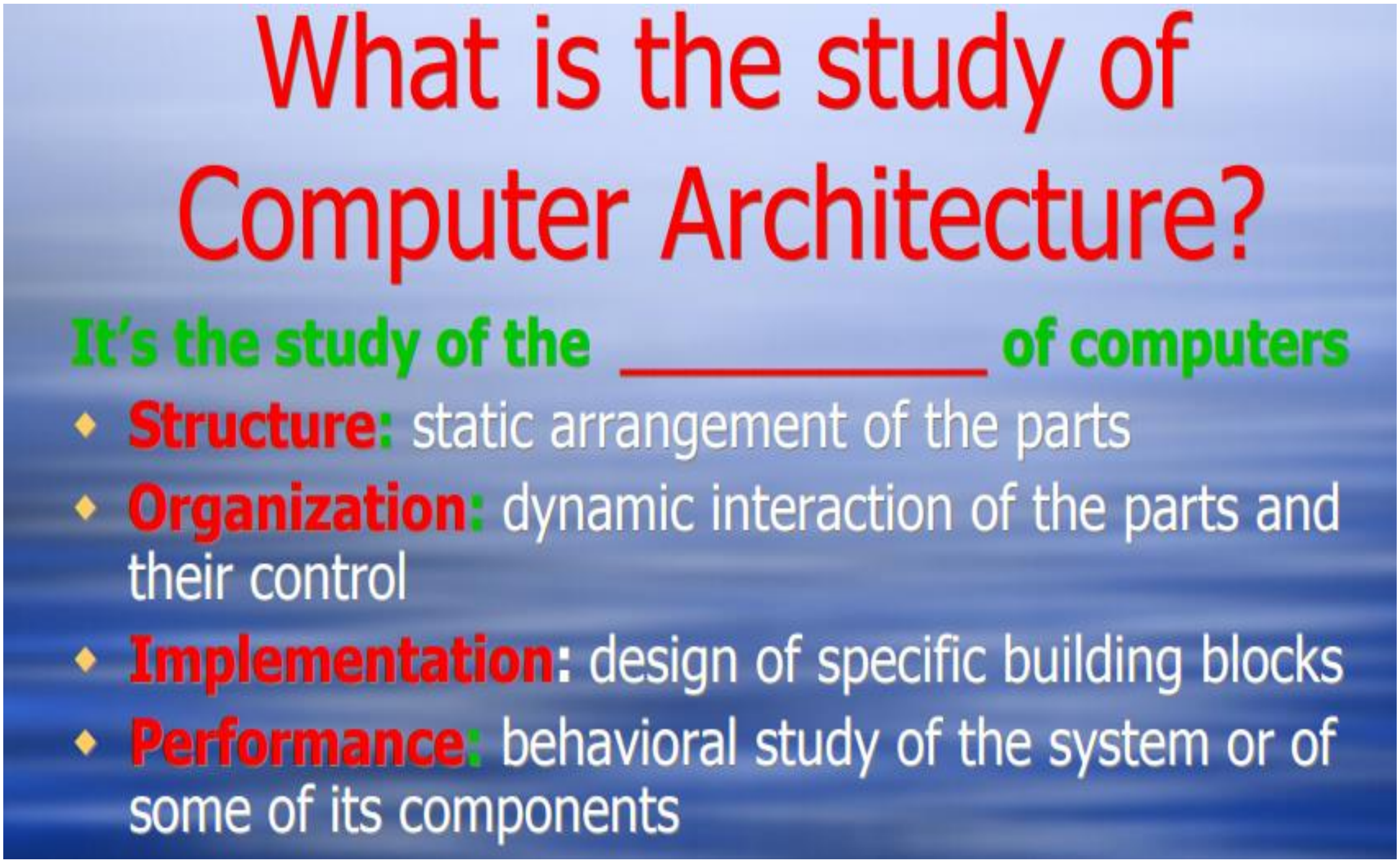
HDL simulators

Grading

- **Homework's**
- **Assignments**
- **Quizzes**
- **2-Midterm Exams**
- **Final Exam**

Course Syllabus

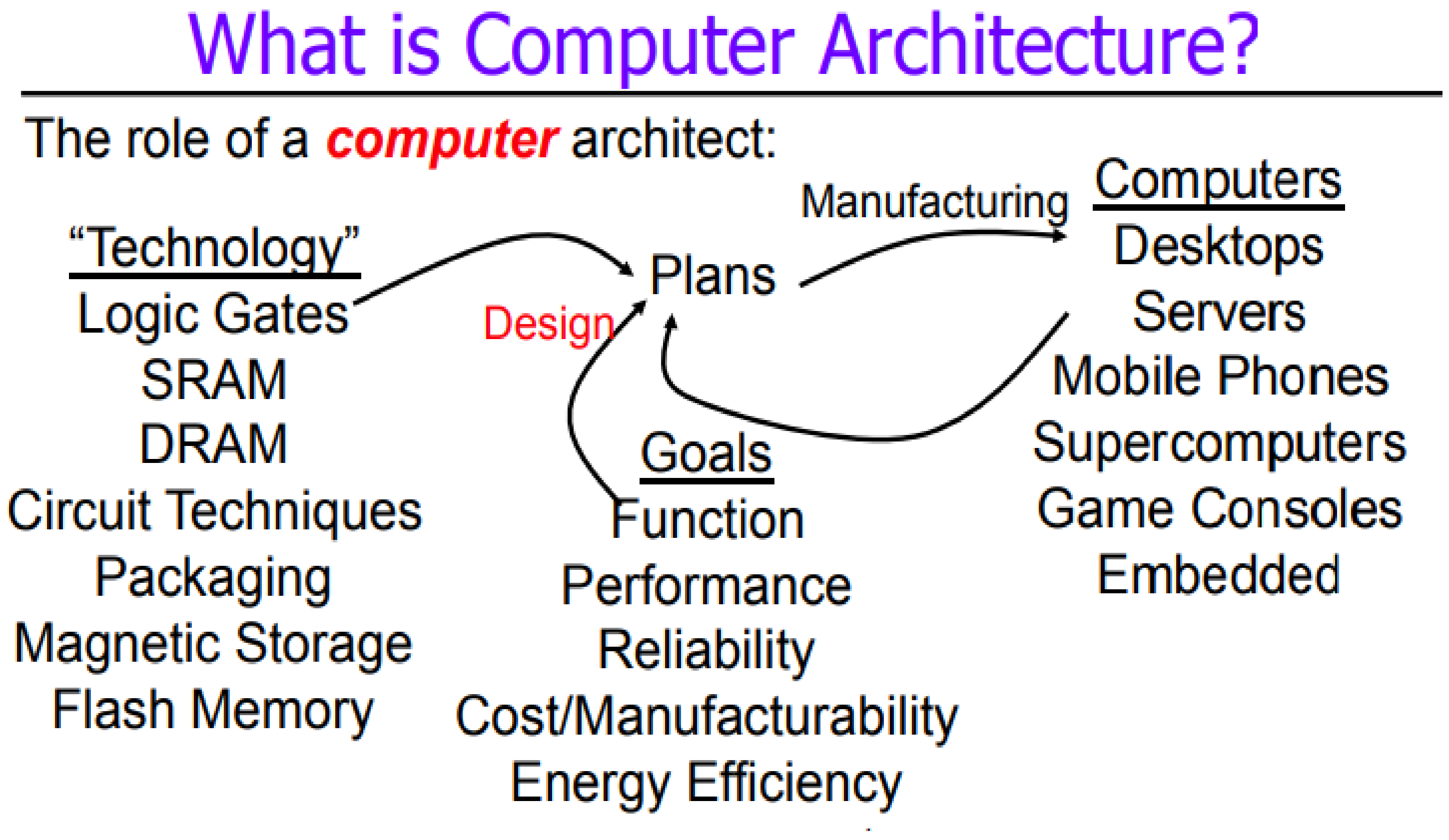
- **CH_1 Computer Abstractions and Technology**
- **(CH_2) Describe the instruction set architecture of a MIPS processor. Analyze, write, and test MIPS assembly language programs.**
- **(CH_3) Describe organization/operation of integer & floating-point units.**
- **(CH_4) Design the datapath and control of a single-cycle, Multi-cycle, and pipelined CPUs, & handle hazards.**
- **(CH_5) Describe the organization/operation of cache memory.**



What is the study of Computer Architecture?

It's the study of the ____________ of computers

- **Structure:** static arrangement of the parts
- **Organization:** dynamic interaction of the parts and their control
- **Implementation:** design of specific building blocks
- **Performance:** behavioral study of the system or of some of its components



What is Computer Architecture?

The role of a ***computer*** architect:

"Technology"
Logic Gates
SRAM
DRAM
Circuit Techniques
Packaging
Magnetic Storage
Flash Memory

Design

Plans

Manufacturing

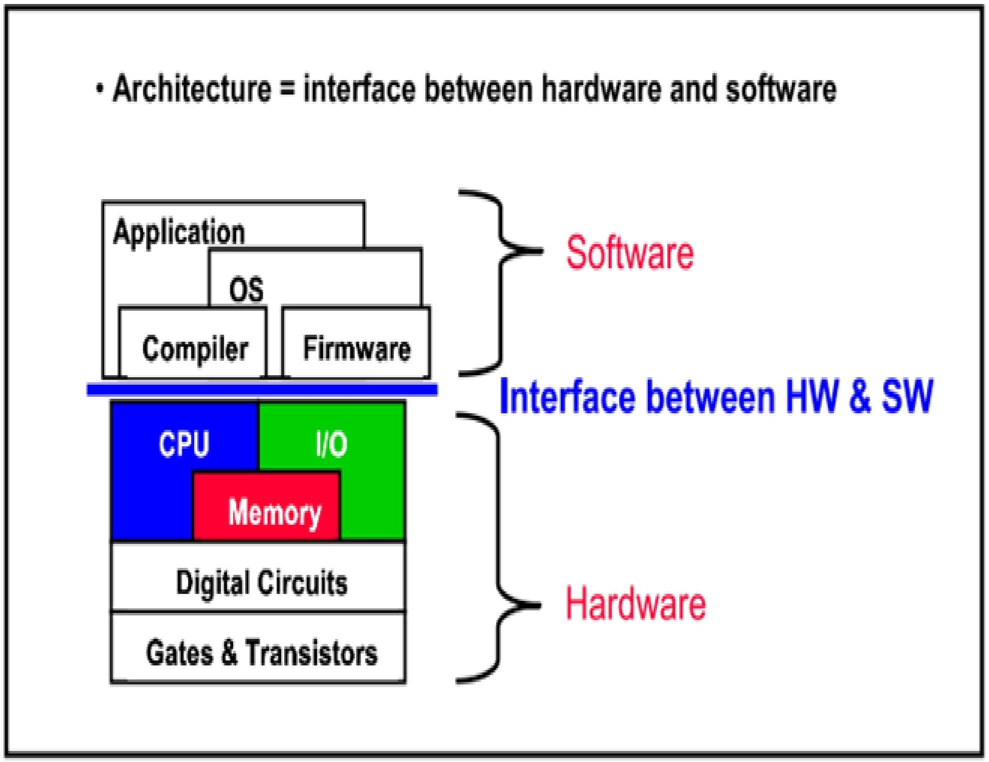
Goals
Function
Performance
Reliability
Cost/Manufacturability
Energy Efficiency

Computers
Desktops
Servers
Mobile Phones
Supercomputers
Game Consoles
Embedded

What is Computer Architecture?

Architecture

- abstraction of the hardware for the programmer
  - instruction set architecture
    - instructions:
      - operations
      - operands, addressing the operands
      - how instructions are encoded
    - storage locations for data
      - registers: how many & what they are used for
      - memory: its size & how it is accessed
    - I/O devices & how to access them
    - software conventions:
      - subroutine calls: who saves the registers, which ones are saved
      - passing parameters: in registers? on the stack?
- the interface between the software & hardware



- Architecture = interface between hardware and software

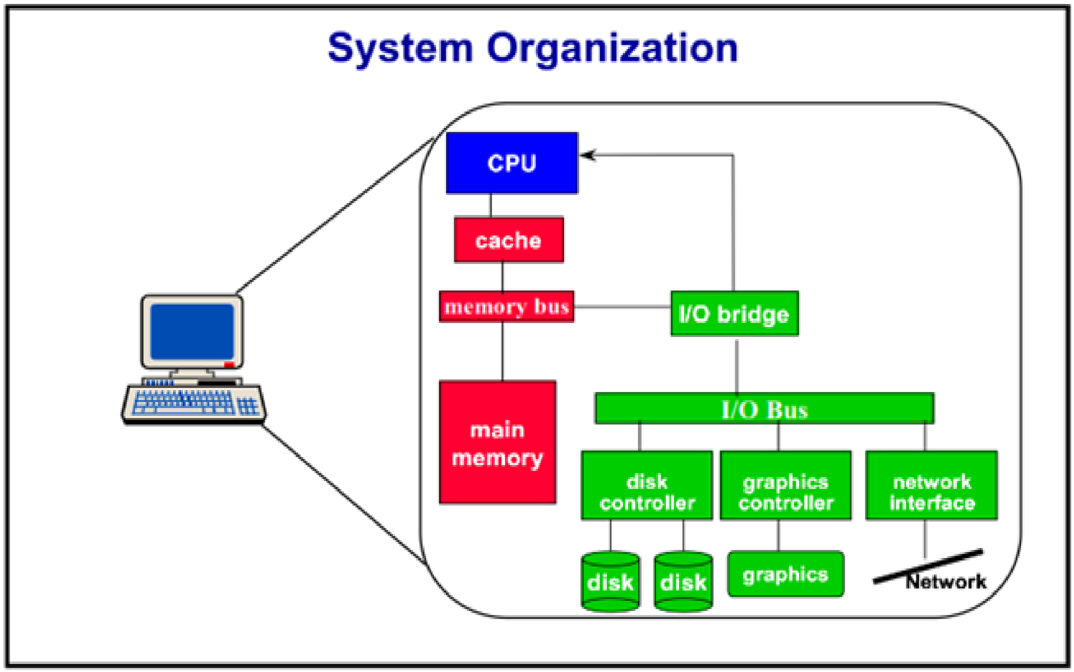
What is Computer Organization?

Organization or Microarchitecture

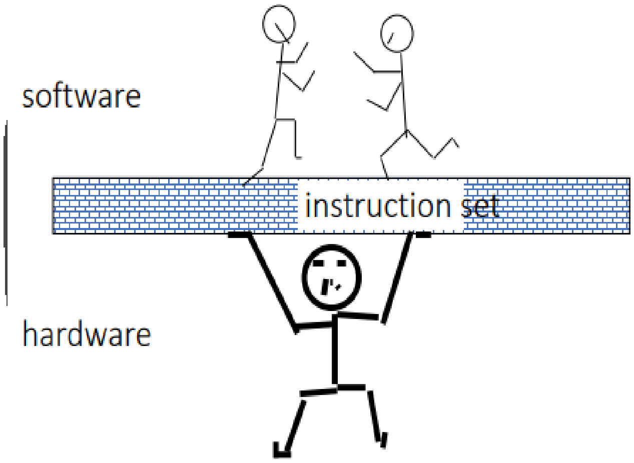
- basic components of a computer
  - on the CPU (ALU, registers, PC, etc.)
  - memory (levels of the cache hierarchy)
- how they operate
- how they are connected together

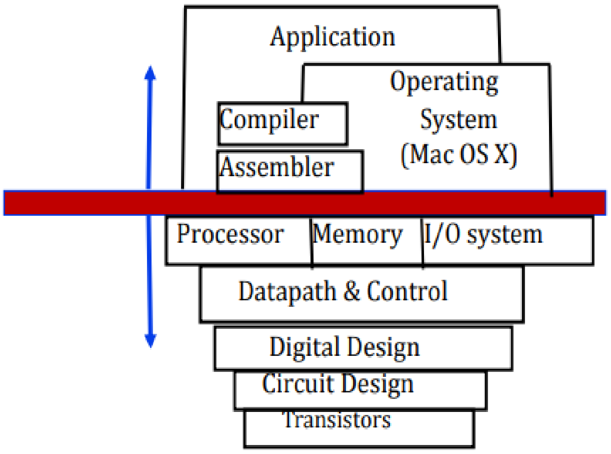
Organization is mostly invisible to the programmer

- today some components are considered ***part of the architecture***
  - why? because a programmer can get better performance if he/ she knows the structure
  - for example: the caches, the pipeline structure

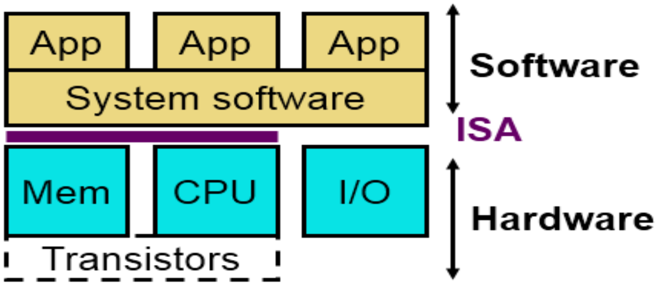


Instruction Set Architecture (ISA)



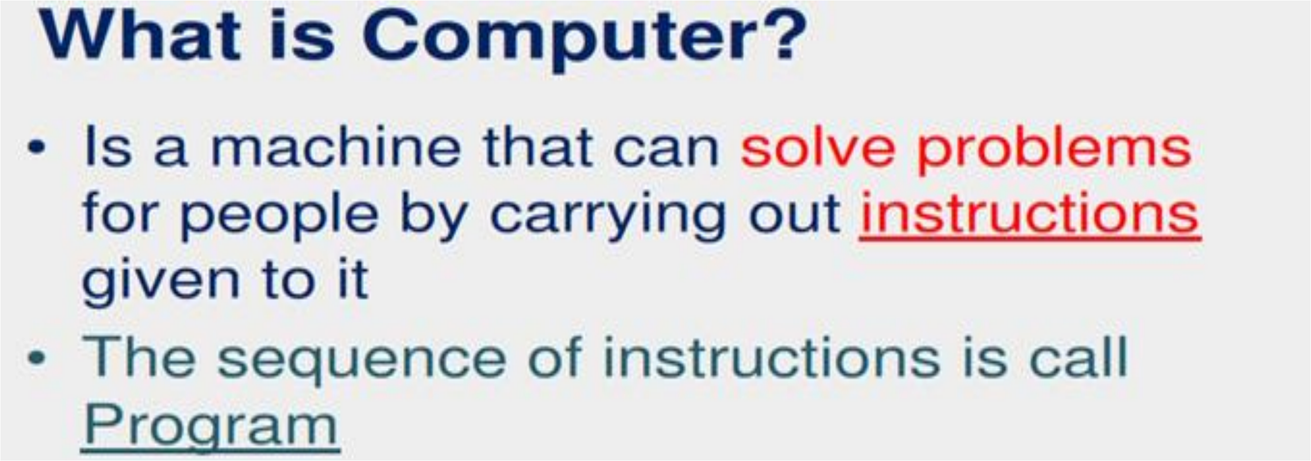


Abstraction, Layering, and Computers



- Computers are complex, built in layers
  - Several **software** layers: assembler, compiler, OS, applications
  - **Instruction set architecture (ISA)**
  - Several **hardware** layers: transistors, gates, CPU/Memory/IO

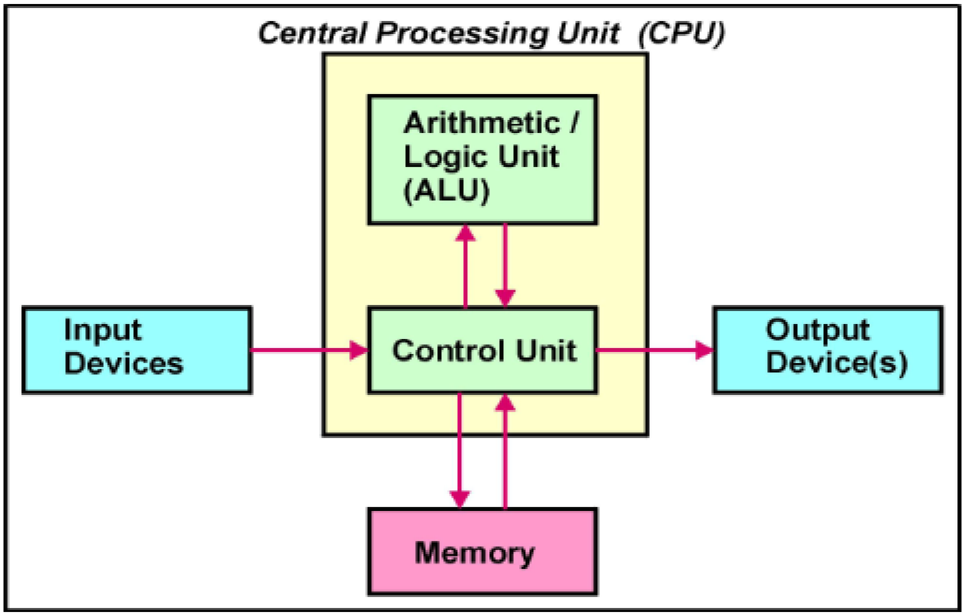
- Build computer bottom up by raising level of abstraction
- Solid-state semi-conductor materials → transistors
- Transistors → gates
- Gates → digital logic elements: latches, muxes, adders
  - Key insight: number representation
- Logic elements → datapath + control = processor



What is Computer?

- Is a machine that can solve problems for people by carrying out instructions given to it
- The sequence of instructions is call Program

The following block diagram describes the Basic Architecture of a Digital Computer:



Applications

■ Automatic teller machines

■ Computers in automobiles

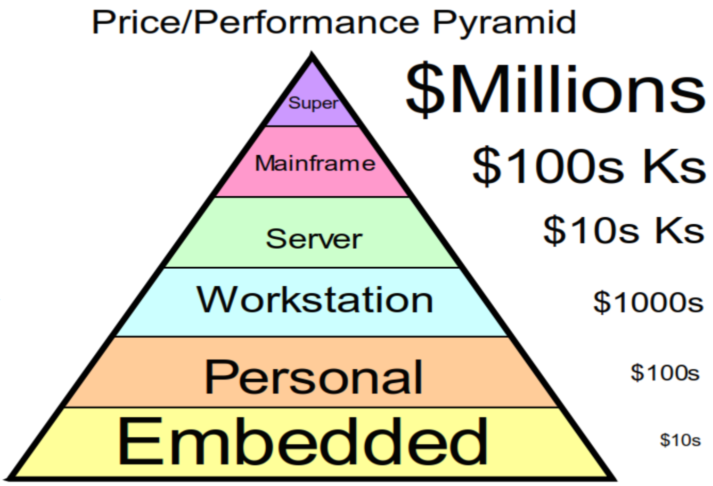
■ Laptop computers

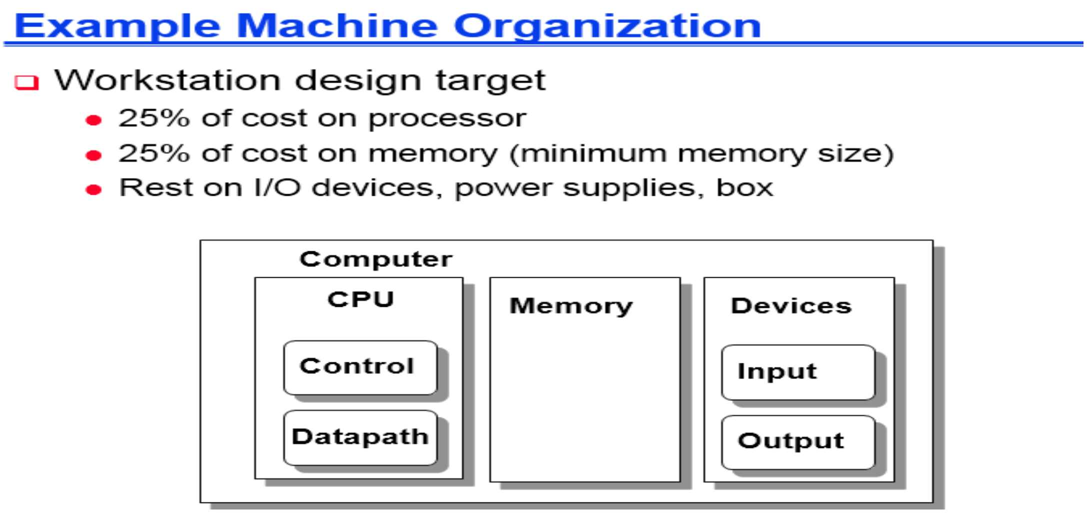
■ Human genome project

■ World Wide Web

Differences Between Computers

- ❖ We have different computers for different purposes.
- ❖ Some can achieve performance needed for high performance gaming
  - E.g., Cell Processor in PlayStation 4.
- ❖ Others can achieve decent enough performance for laptop without using too much power.
  - E.g., Intel Pentium M (for Mobile)
- ❖ Some are cheap enough for your DVD player.
- ❖ And yet others can function reliably enough to be trusted with the control of your car's brakes.





Example Machine Organization

- ❑ Workstation design target
  - 25% of cost on processor
  - 25% of cost on memory (minimum memory size)
  - Rest on I/O devices, power supplies, box

Computer

CPU

Control

Datapath

Memory

Devices

Input

Output

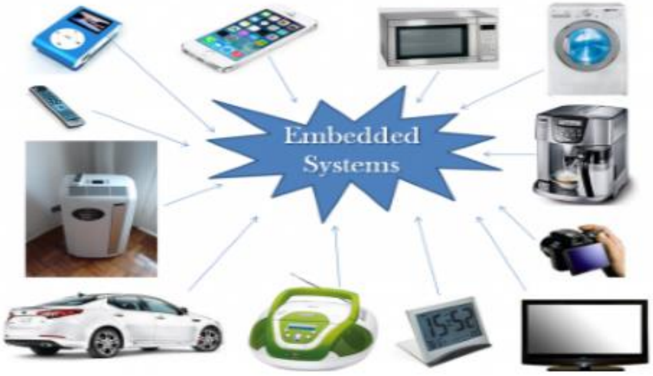
Classes of Computing Applications and Their Characteristics

- Personal computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff

- Server computers
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to building sized

- Supercomputers
  - High-end scientific and engineering calculations
  - Highest capability but represent a small fraction of the overall computer market

- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints



The PostPC Era

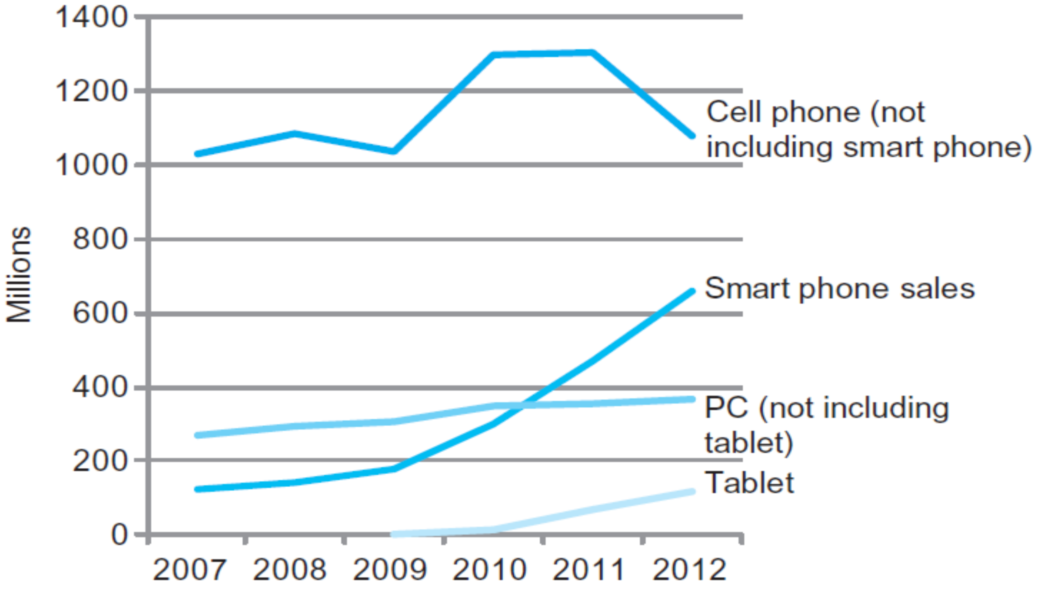
Personal Mobile Device (PMD): are small wireless devices to connect to the Internet

- Battery operated
- Connects to the Internet
- Hundreds of dollars
- Smart phones, tablets.

Cloud computing: refers to large collections of servers that provide services over the Internet.

- Warehouse Scale Computers (WSC)
- Software as a Service (SaaS)
- Portion of software run on a PMD and a portion run in the Cloud
- Amazon and Google

- The number manufactured per year of tablets and smart phones, which reflect the PostPC era, versus personal computers and traditional cell phones. Smart phones represent the recent growth in the cell phone industry, and they passed PCs in 2011. Tablets are the fastest growing category, nearly doubling between 2011 and 2012. Recent PCs and traditional cell phone categories are relatively flat or declining.

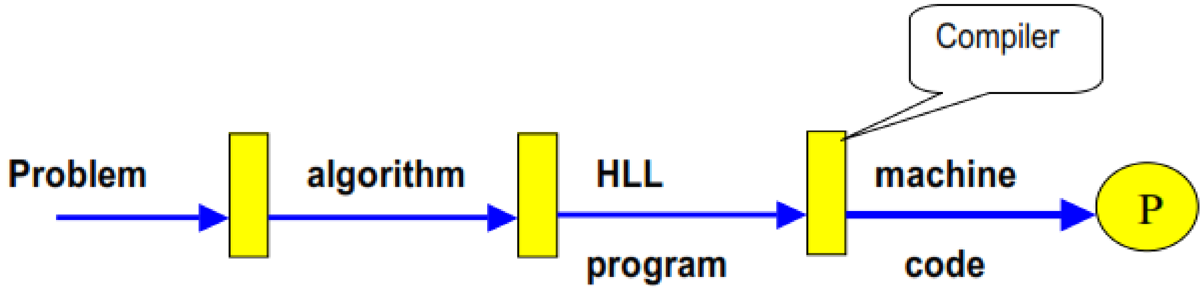


Understanding Performance

Both the software and hardware affect the performance of a program

- Algorithm
  - Determines number of operations executed
- Programming language, compiler, architecture
  - Determine number of machine instructions executed per operation
- Processor and memory system
  - Determine how fast instructions are executed
- I/O system (including OS)
  - Determines how fast I/O operations are executed

Factors influencing computer performance



How fast can you solve a problem on a machine?

Depends on

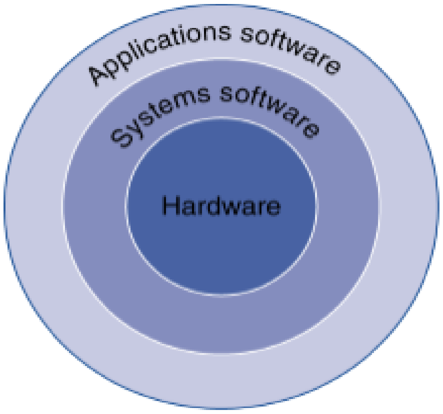
- The algorithm used
- The HLL program code
- The efficiency of the compiler

Hardware and Software as Hierarchical Layers

- A typical application, such as a word processor or a large database system, may consist of millions of lines of code.

- The hardware in a computer can only execute extremely simple low-level instructions.

- To go from a complex application to the simple instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions. (abstraction)!!



- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

Compilers: Translation of a program written in a high-level language, such as C, C++, Java, or Visual Basic into instructions that the hardware can execute.

Operating system: Interfaces between a user's program and the hardware and provides a variety of services and supervisory functions.

From a High-Level Language to the Language of Hardware

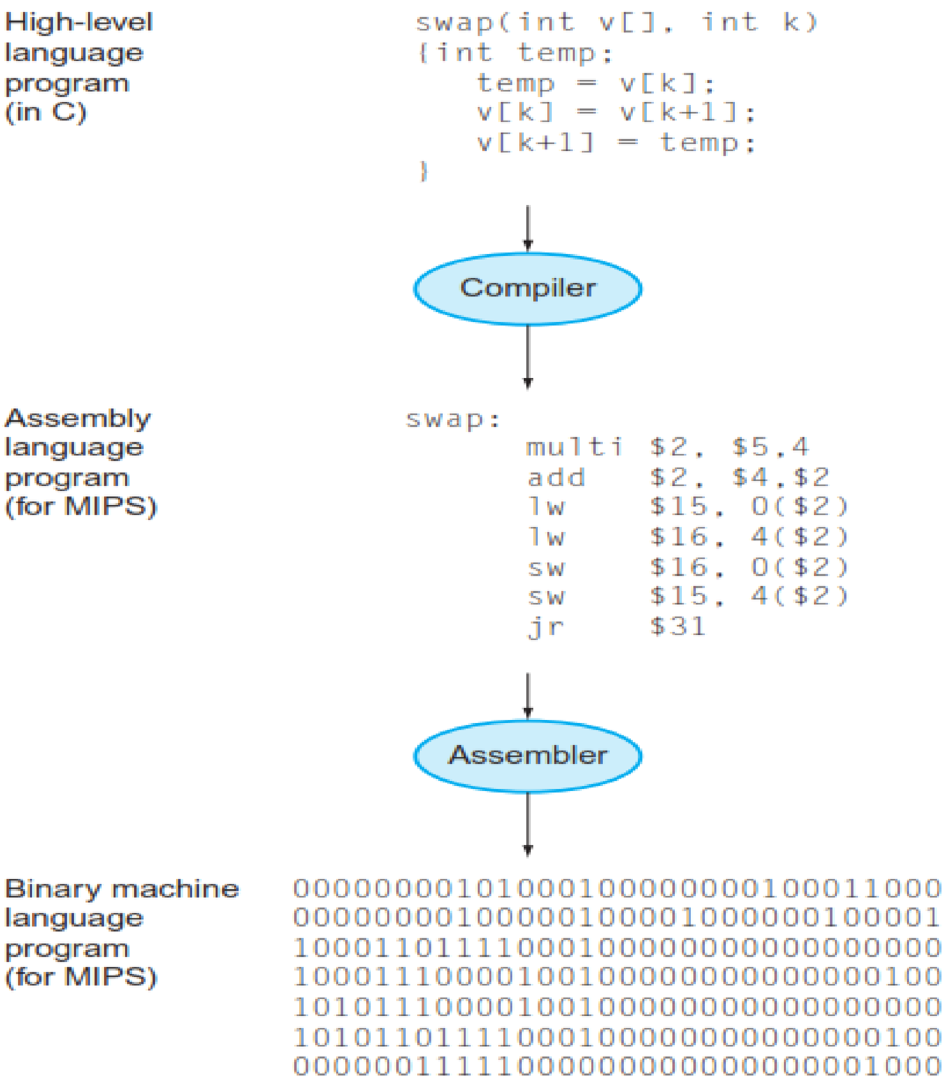
Instruction: A command that computer hardware understands and obeys.

High-level programming language: A portable language such as C, C++,Java, or Visual Basic that is composed of words and algebraic notation that can be translated by a compiler into assembly language.

Assembler: A program that translates a symbolic version of instructions into the binary version.

Assembly Language: A symbolic representation of machine instructions.

Machine Language: A binary representation of machine instructions.


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

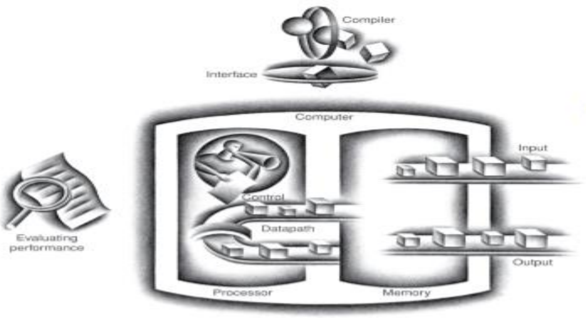
```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

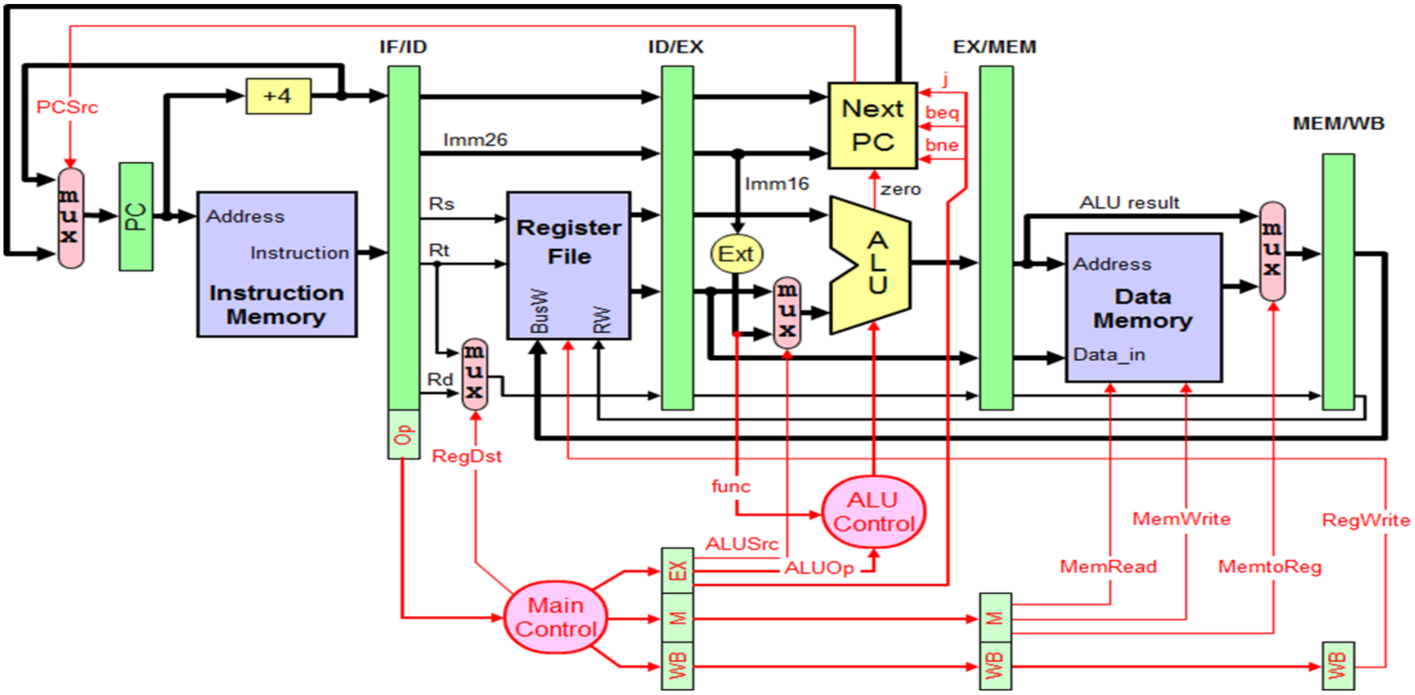
Components of a Computer

The BIG Picture



- Same components for all kinds of computer
  - Desktop, server, embedded
- Input/output includes
  - User-interface devices
    - Display, keyboard, mouse
  - Storage devices
    - Hard disk, CD/DVD, flash
  - Network adapters
    - For communicating with other computers

Inside the Processor (CPU)



Datapath: performs the arithmetic operations.

Control: tells the datapath, memory, and I/O devices what to do according to the wishes of the instructions of the program.

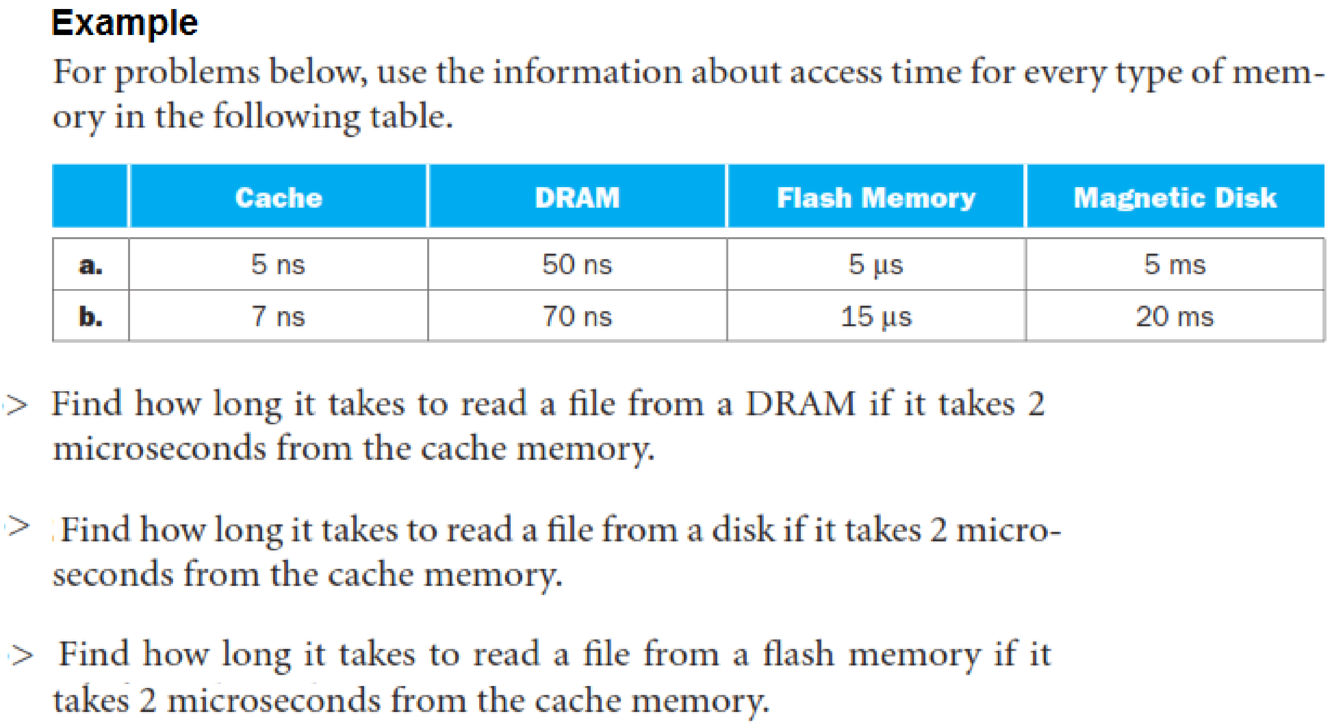
Cache memory A small, fast memory that acts as a buffer for a slower, larger memory.

A Safe Place for Data

- Volatile main memory
  - Loses instructions and data when power off
- Non-volatile secondary memory
  - Magnetic disk
  - Flash memory
  - Optical disk (CDROM, DVD)

Networks

- Communication, resource sharing, nonlocal access
- Local area network (LAN): Ethernet
- Wide area network (WAN): the Internet
- Wireless network: WiFi, Bluetooth



Example

For problems below, use the information about access time for every type of memory in the following table.

	Cache	DRAM	Flash Memory	Magnetic Disk
a.	5 ns	50 ns	5 µs	5 ms
b.	7 ns	70 ns	15 µs	20 ms

> Find how long it takes to read a file from a DRAM if it takes 2 microseconds from the cache memory.

> Find how long it takes to read a file from a disk if it takes 2 microseconds from the cache memory.

> Find how long it takes to read a file from a flash memory if it takes 2 microseconds from the cache memory.

For configuration (a):

* From the table, we find that the DRAM time is equal the 10 * Cache time, so

- The required time to read from DRAM =10*2 microsecond = 20 microsecond.

* From the table, we find that the flash time is equal the 1000 * Cache time, so

- The required time to read from flash =1000*2 microsecond = 2 msec .

* From the table, we find that the magnetic disk time is equal the 1000000 * Cache time, so

- The required time to read from Magnetic Disk =1,000,000 *2 =2 sec

For configuration (b):

* From the table , we find that the DRAM time is equal the 10 * Cache time, so

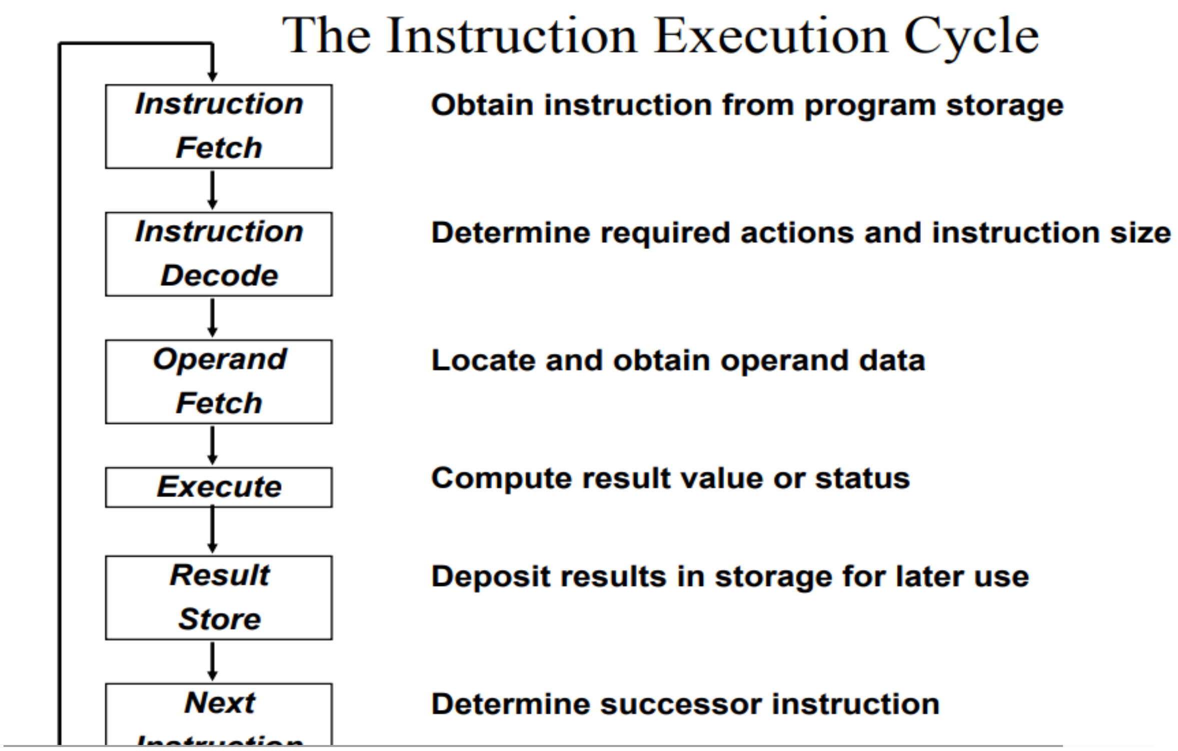
- The required time to read from DRAM =10*2 microsecond = 20 microsecond.

* From the table ,we find that the flash time is equal the 2141 * Cache time, so

- The required time to read from flash =2142*2 microsecond = 4.28 msec.

* From the table ,we find that the magnetic disk time is equal the 2857142 * Cache time, so

- The required time to read from Magnetic Disk =2857142 *2 =5.7 sec



The Instruction Execution Cycle

Instruction Fetch — Obtain instruction from program storage

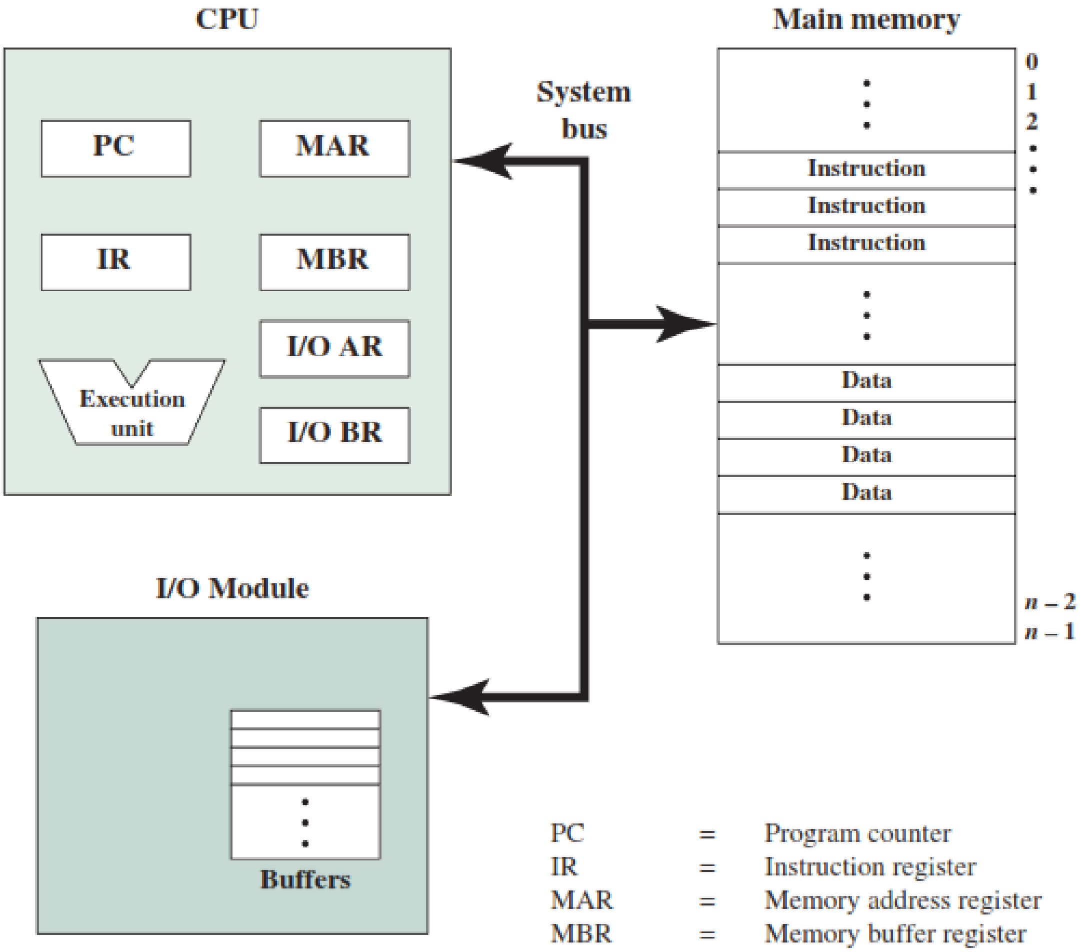
Instruction Decode — Determine required actions and instruction size

Operand Fetch — Locate and obtain operand data

Execute — Compute result value or status

Result Store — Deposit results in storage for later use

Next Instruction — Determine successor instruction



PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

Technology Trends

- Electronics technology continues to evolve
  - Increased capacity and performance
  - Reduced cost

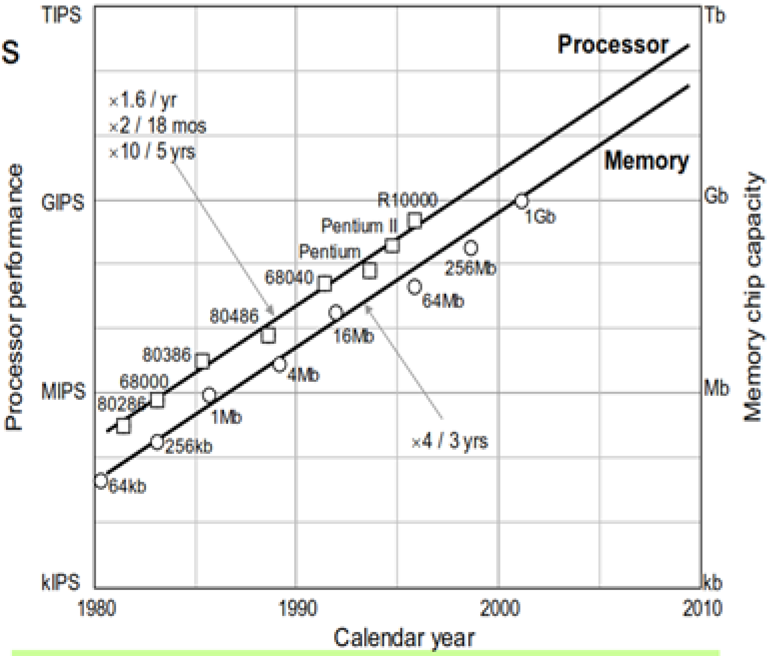
Year	Technology	Relative performance/cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit (IC)	900
1995	Very large scale IC (VLSI)	2,400,000
2013	Ultra large scale IC	250,000,000,000

Moore' s Law

- In 1965, Gordon Moore noted that the number of transistors on a chip doubled every 18 to 24 months.

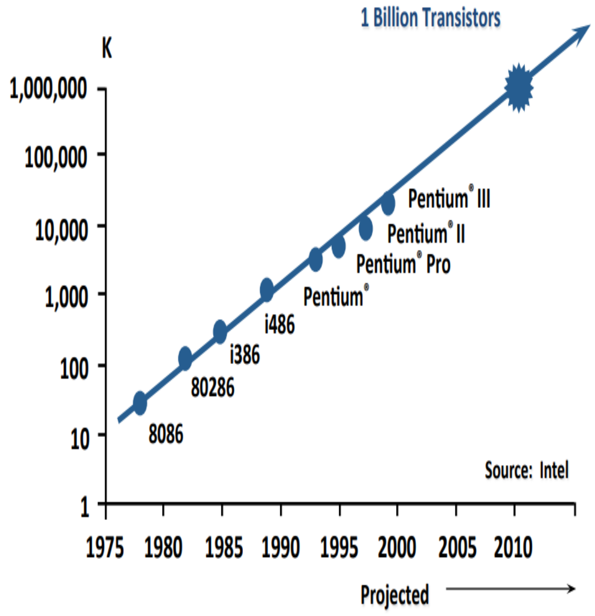
- He made a prediction that semiconductor technology will double its effectiveness every 18 months

Moore's Law

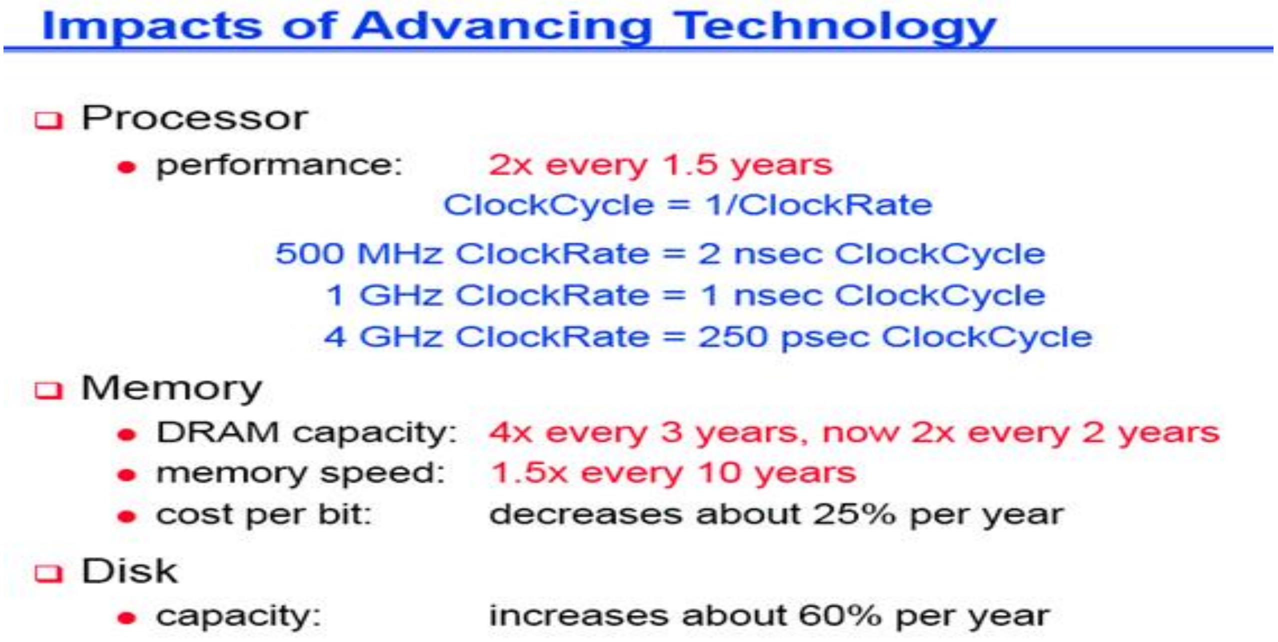


Trends in processor performance and DRAM memory chip capacity (Moore's law).

Transistor Counts



Source: Intel



Impacts of Advancing Technology

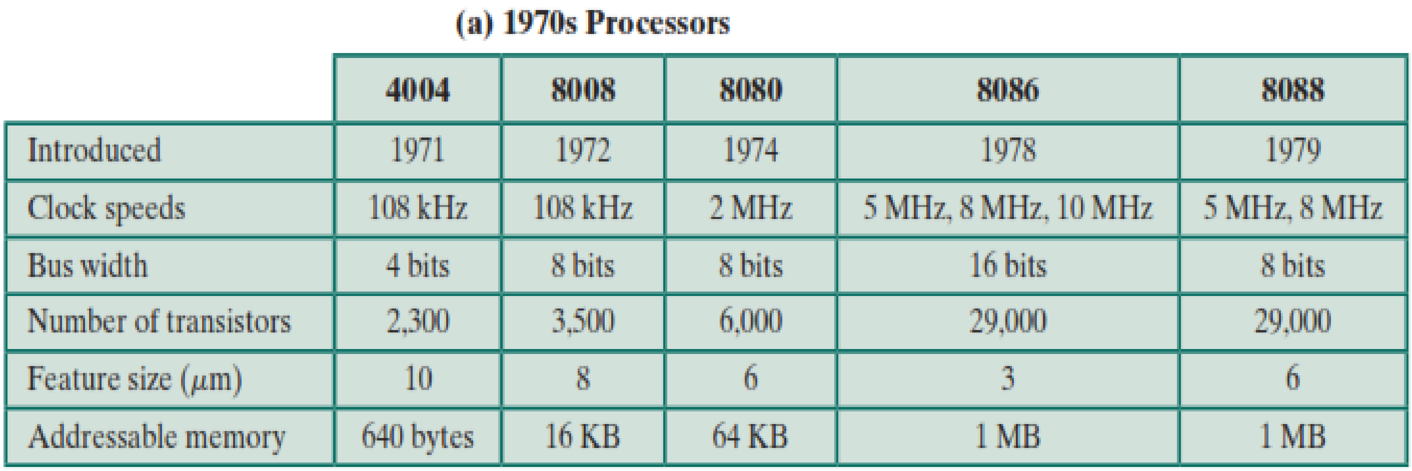
- Processor
  - performance: 2x every 1.5 years

    ClockCycle = 1/ClockRate

    500 MHz ClockRate = 2 nsec ClockCycle

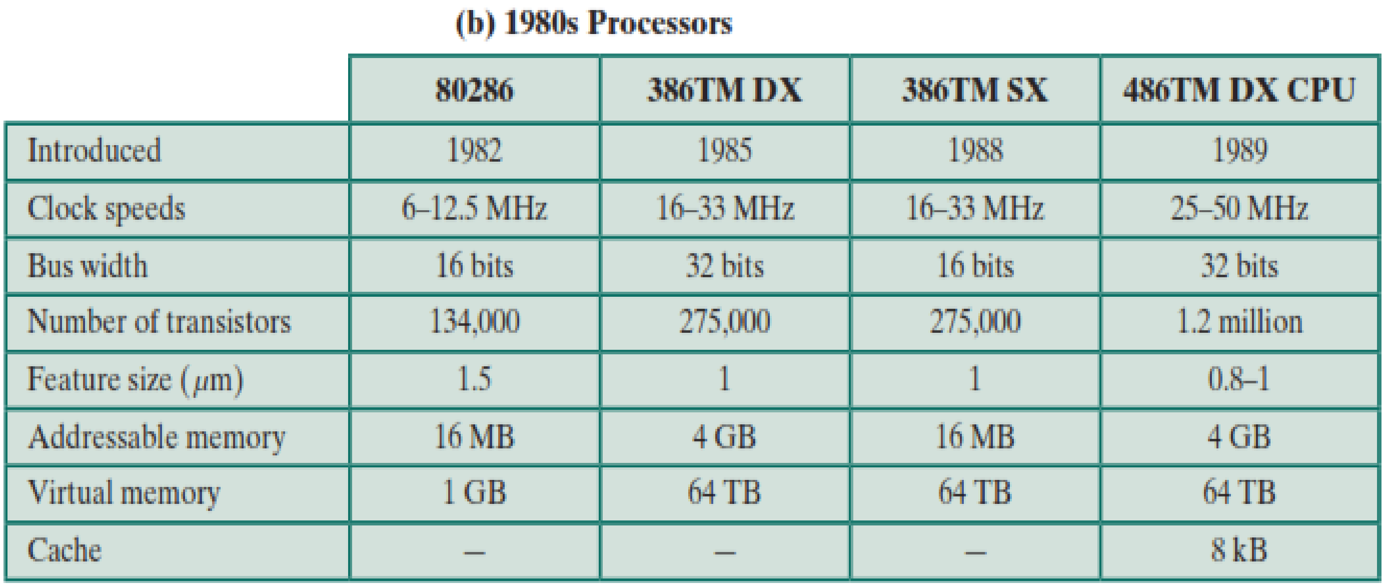
    1 GHz ClockRate = 1 nsec ClockCycle

    4 GHz ClockRate = 250 psec ClockCycle
- Memory
  - DRAM capacity: 4x every 3 years, now 2x every 2 years
  - memory speed: 1.5x every 10 years
  - cost per bit: decreases about 25% per year
- Disk
  - capacity: increases about 60% per year



(a) 1970s Processors

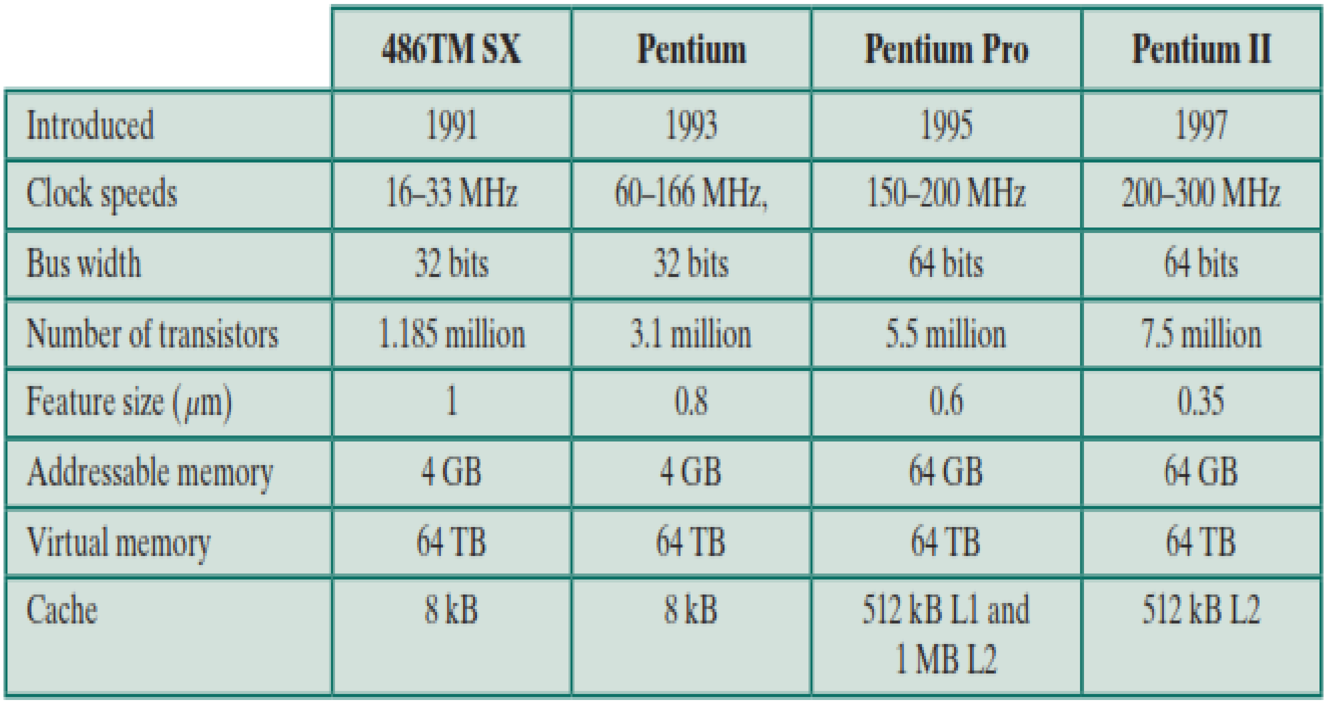
	4004	8008	8080	8086	8088
Introduced	1971	1972	1974	1978	1979
Clock speeds	108 kHz	108 kHz	2 MHz	5 MHz, 8 MHz, 10 MHz	5 MHz, 8 MHz
Bus width	4 bits	8 bits	8 bits	16 bits	8 bits
Number of transistors	2,300	3,500	6,000	29,000	29,000
Feature size (μm)	10	8	6	3	6
Addressable memory	640 bytes	16 KB	64 KB	1 MB	1 MB



(b) 1980s Processors

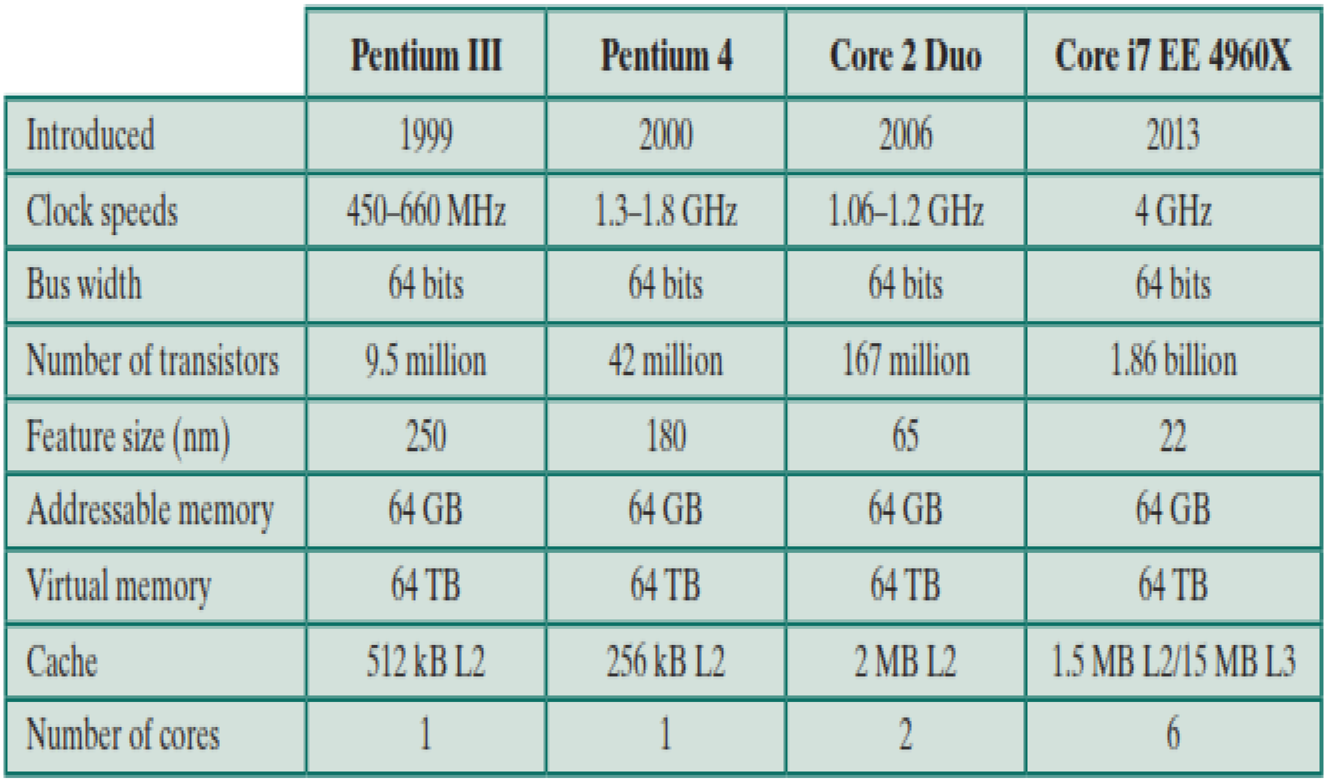
	80286	386TM DX	386TM SX	486TM DX CPU
Introduced	1982	1985	1988	1989
Clock speeds	6–12.5 MHz	16–33 MHz	16–33 MHz	25–50 MHz
Bus width	16 bits	32 bits	16 bits	32 bits
Number of transistors	134,000	275,000	275,000	1.2 million
Feature size (μm)	1.5	1	1	0.8–1
Addressable memory	16 MB	4 GB	16 MB	4 GB
Virtual memory	1 GB	64 TB	64 TB	64 TB
Cache	–	–	–	8 kB

(c) 1990s Processors



	486TM SX	Pentium	Pentium Pro	Pentium II
Introduced	1991	1993	1995	1997
Clock speeds	16–33 MHz	60–166 MHz,	150–200 MHz	200–300 MHz
Bus width	32 bits	32 bits	64 bits	64 bits
Number of transistors	1.185 million	3.1 million	5.5 million	7.5 million
Feature size (μm)	1	0.8	0.6	0.35
Addressable memory	4 GB	4 GB	64 GB	64 GB
Virtual memory	64 TB	64 TB	64 TB	64 TB
Cache	8 kB	8 kB	512 kB L1 and 1 MB L2	512 kB L2

(d) Recent Processors



	Pentium III	Pentium 4	Core 2 Duo	Core i7 EE 4960X
Introduced	1999	2000	2006	2013
Clock speeds	450–660 MHz	1.3–1.8 GHz	1.06–1.2 GHz	4 GHz
Bus width	64 bits	64 bits	64 bits	64 bits
Number of transistors	9.5 million	42 million	167 million	1.86 billion
Feature size (nm)	250	180	65	22
Addressable memory	64 GB	64 GB	64 GB	64 GB
Virtual memory	64 TB	64 TB	64 TB	64 TB
Cache	512 kB L2	256 kB L2	2 MB L2	1.5 MB L2/15 MB L3
Number of cores	1	1	2	6

Semiconductor Technology

- **Silicon: semiconductor**
- **With a special chemical process, it is possible to add materials to silicon to transform into one of three devices**
  - **Conductors** (using either microscopic copper or aluminum wire)
  - **Insulators** (like plastic sheathing or glass)
  - **Switch** (Areas that can conduct or insulate under special conditions)
- **VLSI circuit is just billions of combinations of conductors, insulators, and switches manufactured in a single small package.**

Manufacturing ICs

Silicon crystal ingot: A rod composed of a silicon crystal that is between 8 and 12 inches in diameter and about 12 to 24 inches long.

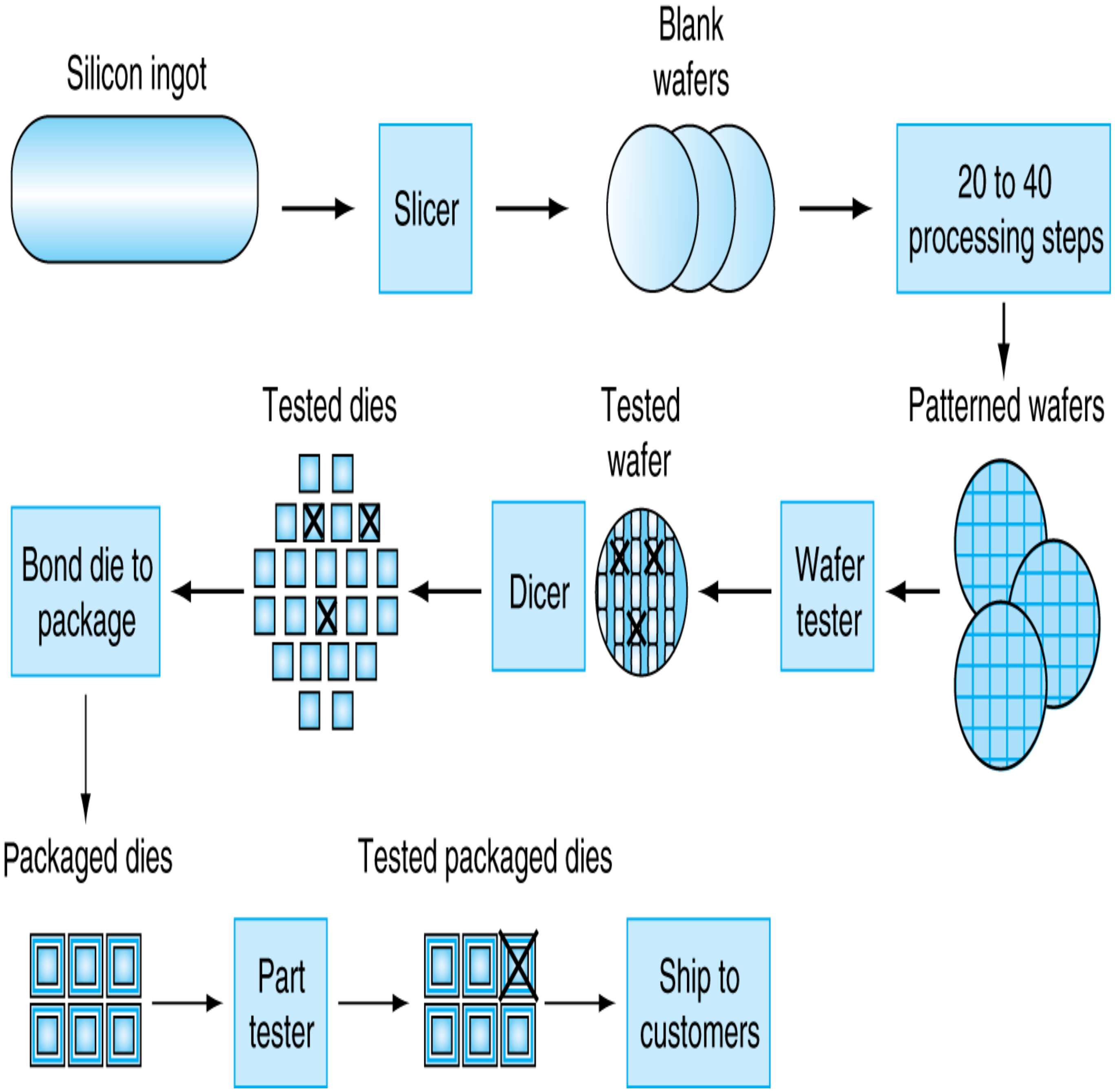
Wafer: A slice from a silicon ingot no more than 0.1 inches thick, used to create chips.

Defect: A microscopic flaw in a wafer or in patterning steps that can result in the failure of the die containing that defect.

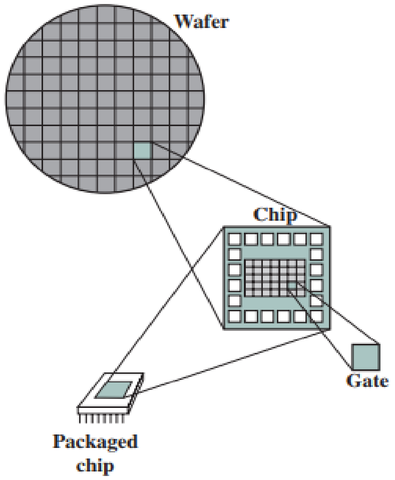
Die: The individual rectangular sections that are cut from a wafer, more informally known as *chips*.

Yield: The percentage of good dies from the total number of dies on the wafer.

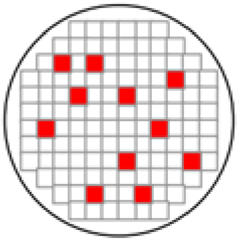
Bonding: connected the good dies to the input/output pins of a package.



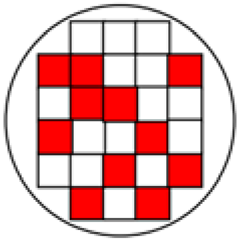
The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers. These patterned wafers are then tested with a wafer tester and a map of the good parts is made. Then, the wafers are diced into dies. The good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers.



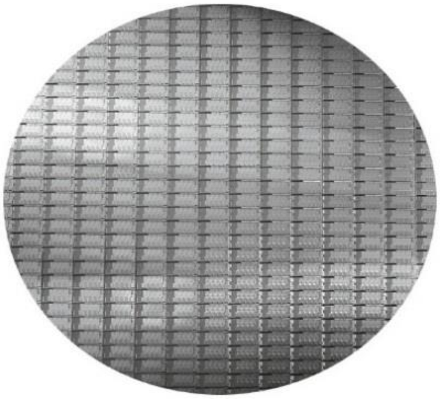
Relationship among Wafer, Chip, and Gate



120 dies, 109 good



26 dies, 15 good



Intel Core i7 Wafer

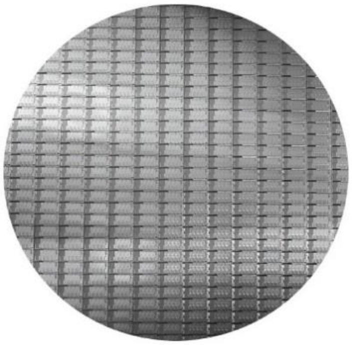
12 inch (300mm)

280 chip

217 mm^2

32 nm technology

731,000,000 transistor



Intel Pentium 4 Wafer

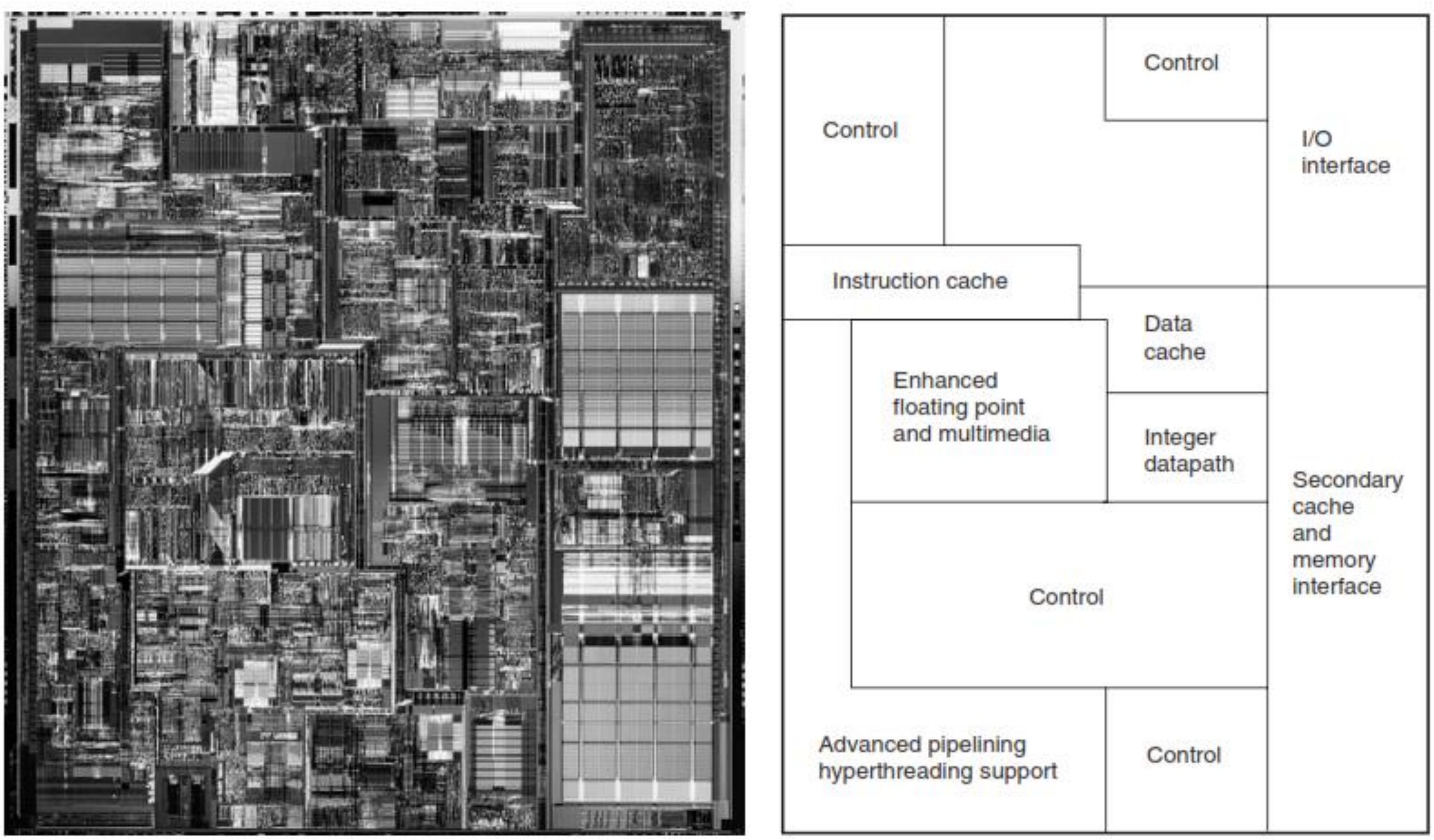
8 inch (200mm)

165 chip

250 mm^2

180 nm technology

55,000,000 transistor


Control
Control
I/O interface
Instruction cache
Data cache
Enhanced floating point and multimedia
Integer datapath
Secondary cache and memory interface
Control
Advanced pipelining hyperthreading support
Control

Inside the processor chip The left-hand side is a microphotograph of the Pentium 4 processor chip, and the right-hand side shows the major blocks in the processor.

Integrated Circuit Cost

The cost of an integrated circuit can be expressed in the following equations:

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

Note

The number of dies per wafer is approximately the **area of the wafer** divided by the **area of the die**. It can be more accurately estimated by

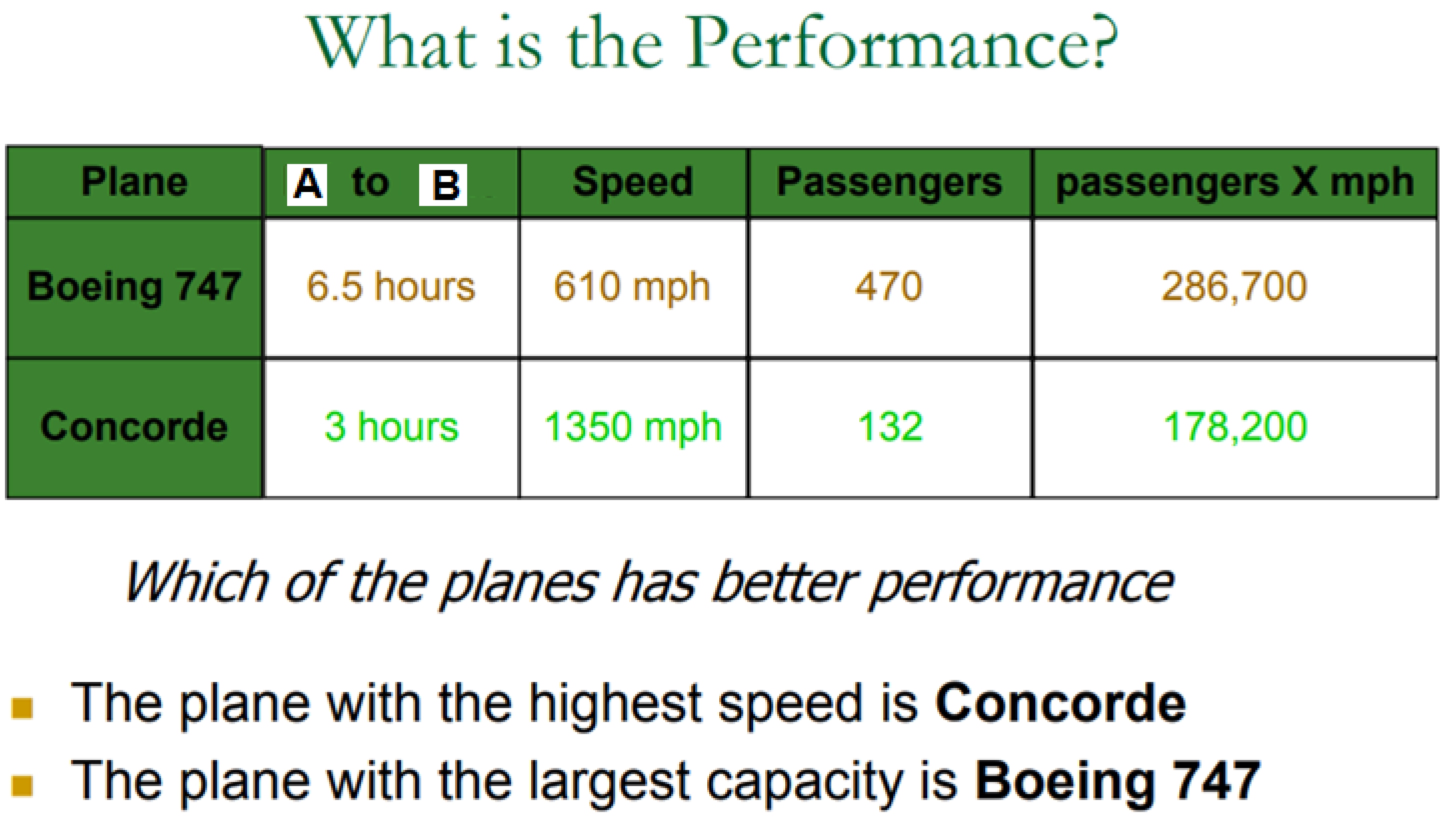
$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2 \times \text{Die area}}}$$

Example Find the number of dies per 300 mm (30 cm) wafer for a die that is 1.5 cm on a side and for a die that is 1.0 cm on a side.

Answer When die area is 2.25 cm^2:

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{2.25} - \frac{\pi \times 30}{\sqrt{2 \times 2.25}} = \frac{706.9}{2.25} - \frac{94.2}{2.12} = 270$$

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{1.00} - \frac{\pi \times 30}{\sqrt{2 \times 1.00}} = \frac{706.9}{1.00} - \frac{94.2}{1.41} = 640$$

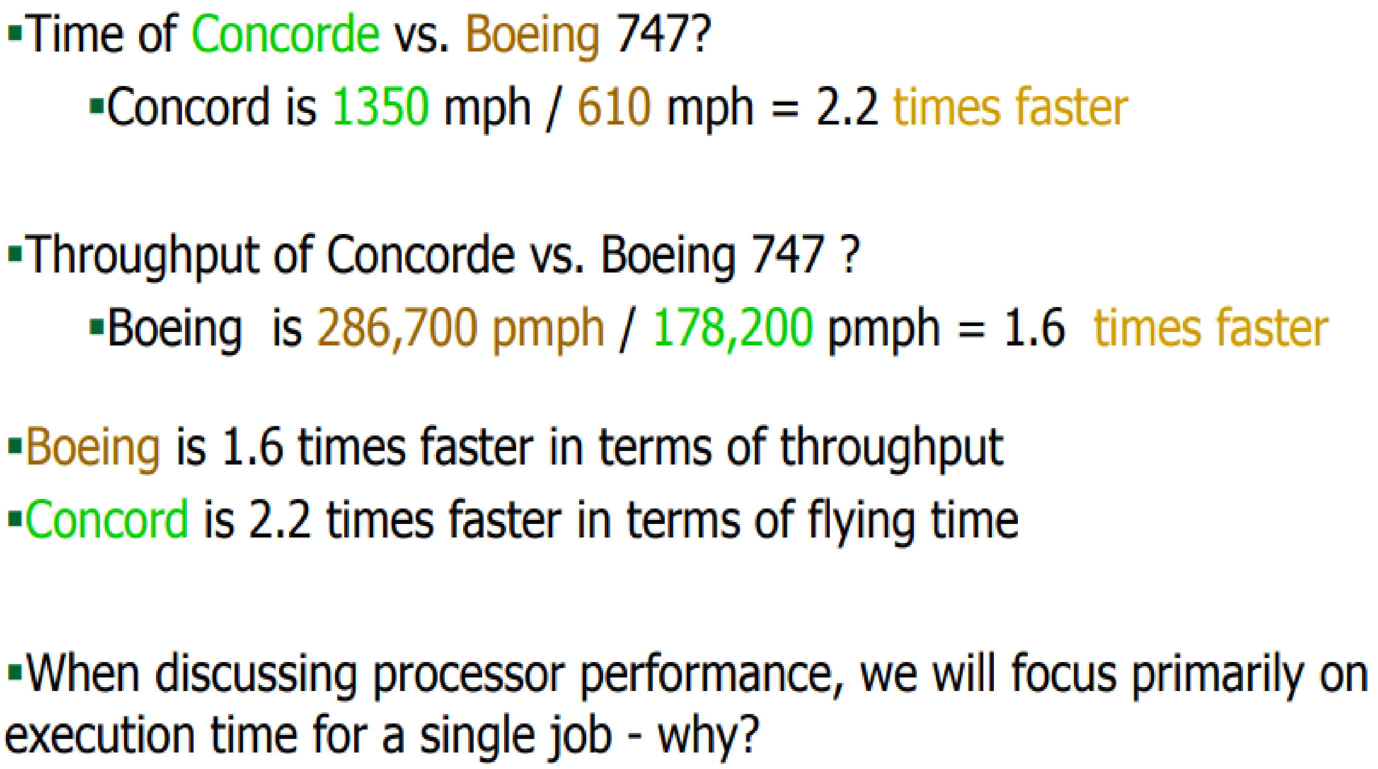


What is the Performance?

Plane	A to B	Speed	Passengers	passengers X mph
Boeing 747	6.5 hours	610 mph	470	286,700
Concorde	3 hours	1350 mph	132	178,200

Which of the planes has better performance

- The plane with the highest speed is **Concorde**
- The plane with the largest capacity is **Boeing 747**



- Time of Concorde vs. Boeing 747?
  - Concord is 1350 mph / 610 mph = 2.2 times faster

- Throughput of Concorde vs. Boeing 747 ?
  - Boeing is 286,700 pmph / 178,200 pmph = 1.6 times faster

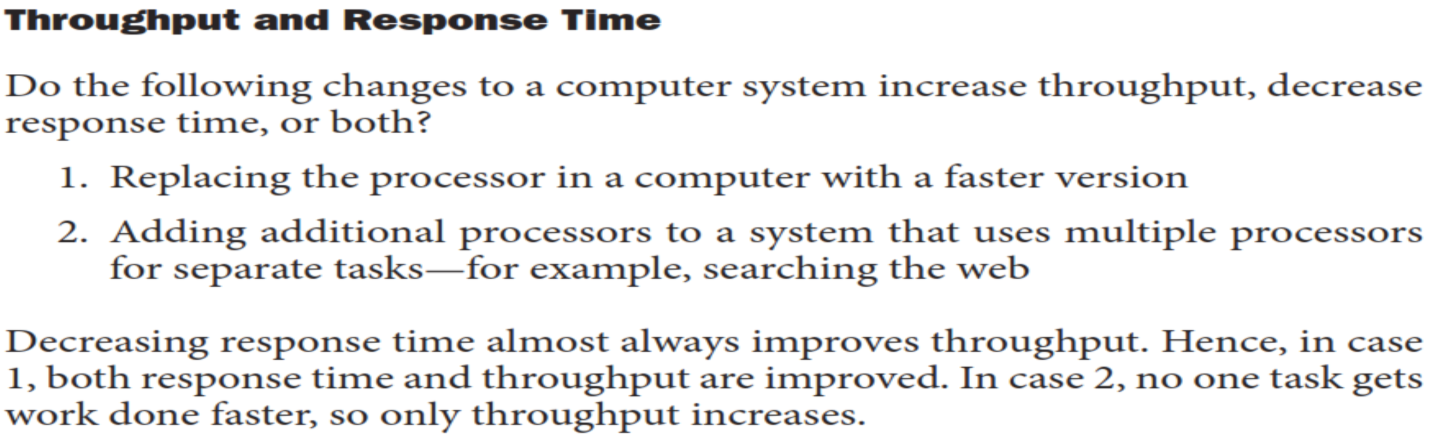
- Boeing is 1.6 times faster in terms of throughput
- Concord is 2.2 times faster in terms of flying time

- When discussing processor performance, we will focus primarily on execution time for a single job - why?

Response Time and Throughput

1- **Response time (Execution time)**: The total time required the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

2- **Throughput (Bandwidth)**: It is the number of tasks completed per unit time.



Throughput and Response Time

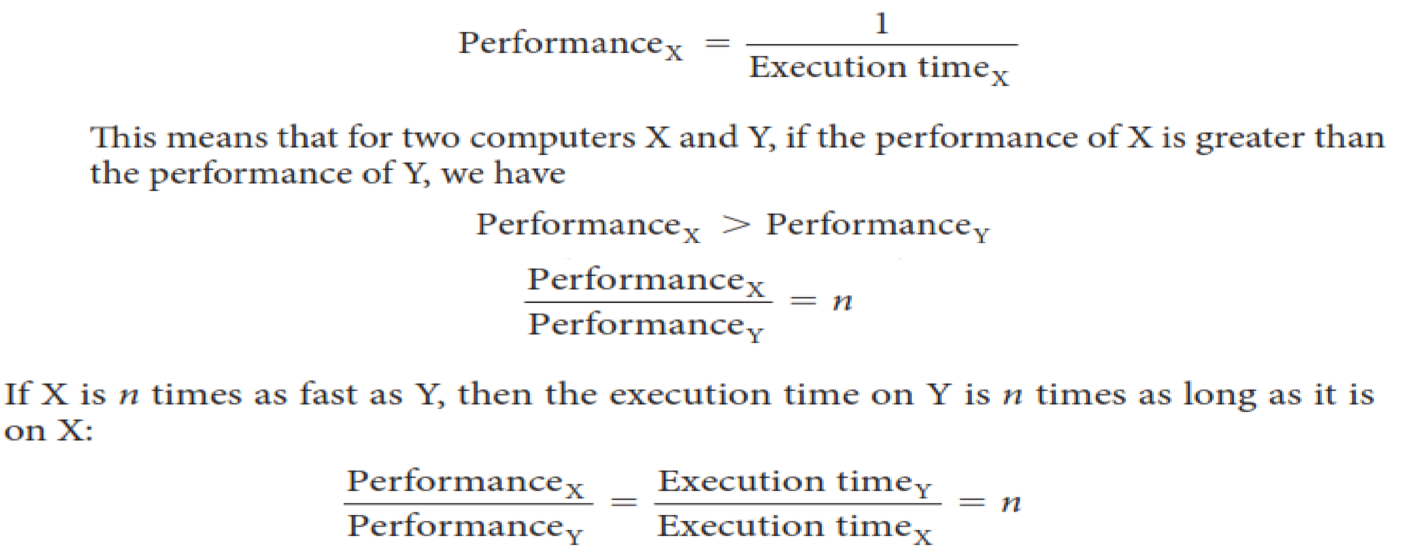
Do the following changes to a computer system increase throughput, decrease response time, or both?

1. Replacing the processor in a computer with a faster version
2. Adding additional processors to a system that uses multiple processors for separate tasks—for example, searching the web

Decreasing response time almost always improves throughput. Hence, in case 1, both response time and throughput are improved. In case 2, no one task gets work done faster, so only throughput increases.

Note: ***In many real computer systems, changing either execution time or throughput often affects the other.***

To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:



$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

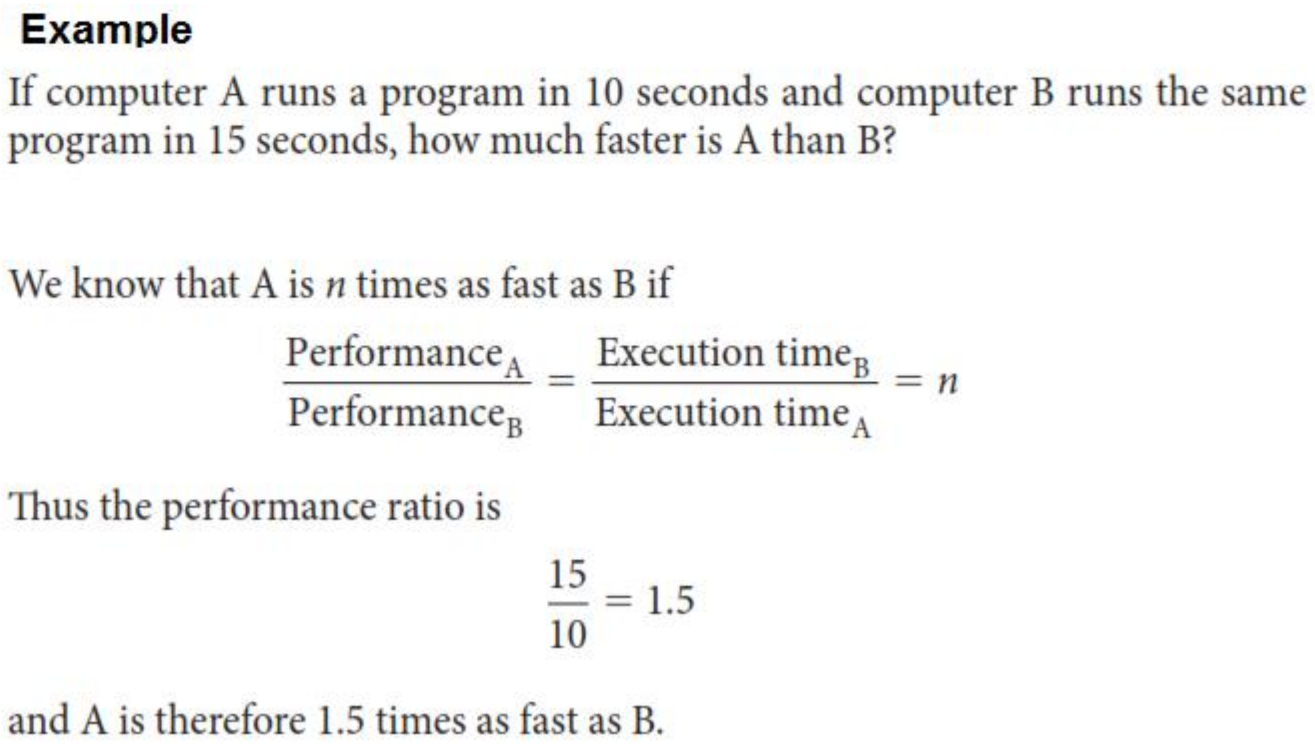
This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is n times as fast as Y, then the execution time on Y is n times as long as it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$



Example

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

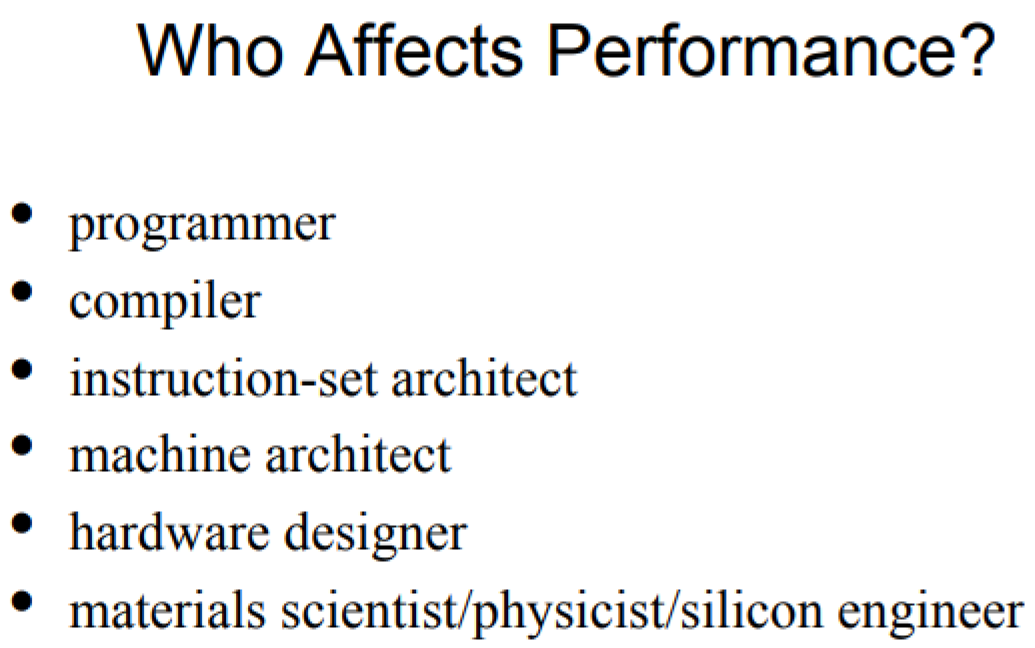
We know that A is n times as fast as B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times as fast as B.



Who Affects Performance?

- programmer
- compiler
- instruction-set architect
- machine architect
- hardware designer
- materials scientist/physicist/silicon engineer

Measuring Performance

1- CPU execution time (CPU time)

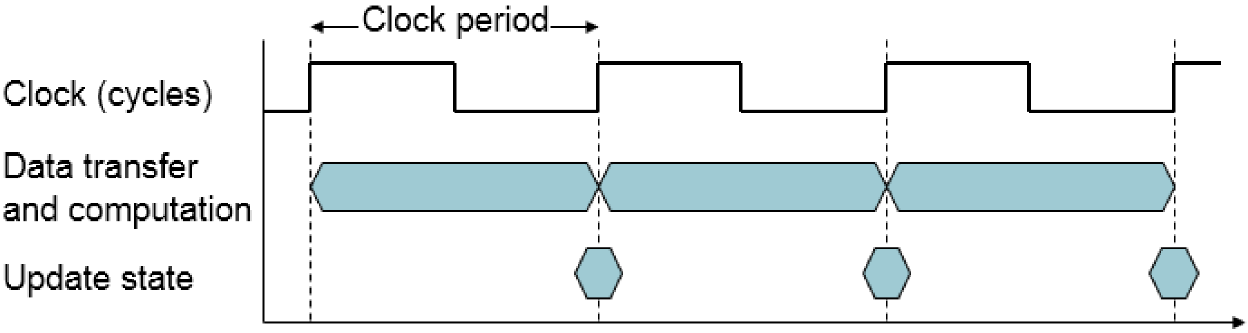
CPU time (or CPU Execution time) is the time between the start and the end of execution of a given program. This time accounts for the time CPU is computing the given program, including operating system routines executed on the program's behalf, and it does not include the time waiting for I/O and running other programs. **CPU time** Comprises ***user CPU time*** and **system CPU time**

- **user CPU time** : The CPU time spent in a program itself.
- **system CPU time** The CPU time spent in the operating system performing tasks on behalf of the program.

2- Elapsed time

Total response time, including all aspects, processing, I/O, OS overhead, idle time.

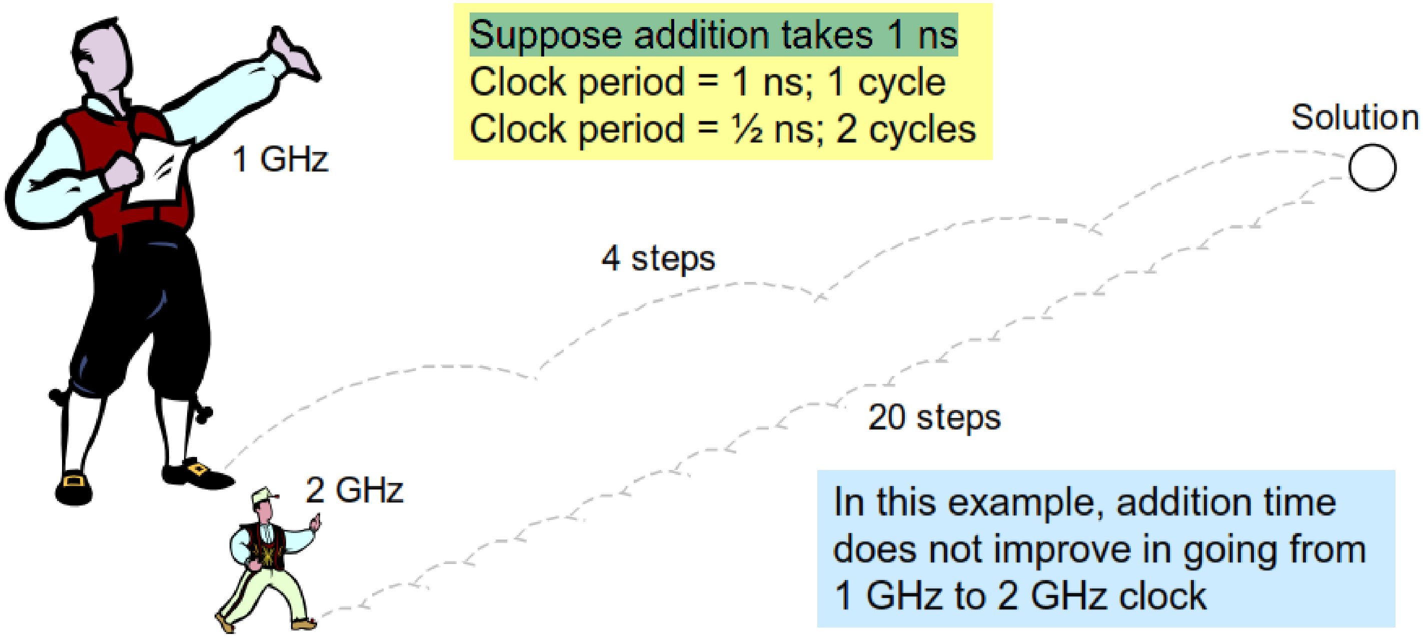
3- CPU Clocking



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

- 1 GHz = 10^9 cycles / s (cycle time 10^{-9} s = 1 ns)
  200 MHz = 200×10^6 cycles / s (cycle time = 5 ns)

Faster Clock $\neq$ Shorter Running Time



Suppose addition takes 1 ns
Clock period = 1 ns; 1 cycle
Clock period = ½ ns; 2 cycles

In this example, addition time does not improve in going from 1 GHz to 2 GHz clock

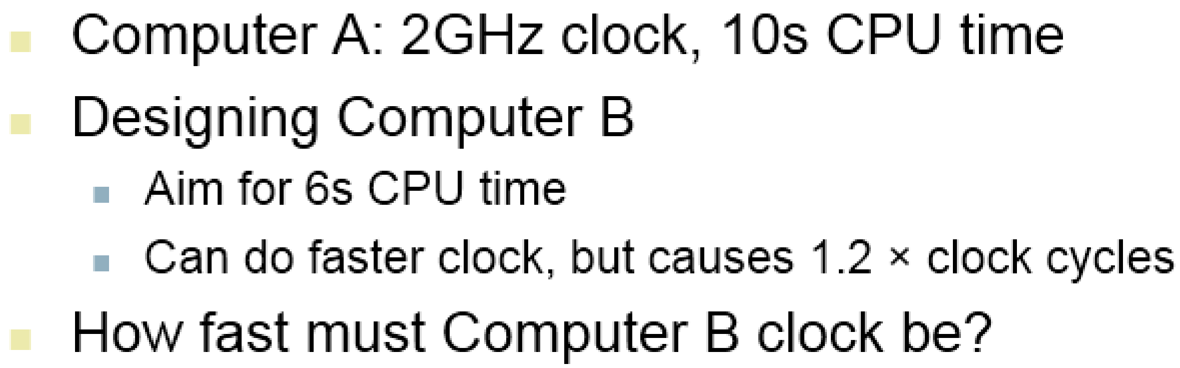
Faster steps do not necessarily mean shorter travel time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}} \quad = \frac{\text{cycle}}{\text{cycle/sec}} = \text{sec}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

Example



- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

Answer

The number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \ \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \ \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \ \text{cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \ \text{cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \ \text{cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \ \text{cycles}}{\text{second}} = \frac{4 \times 10^9 \ \text{cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

Instruction Performance

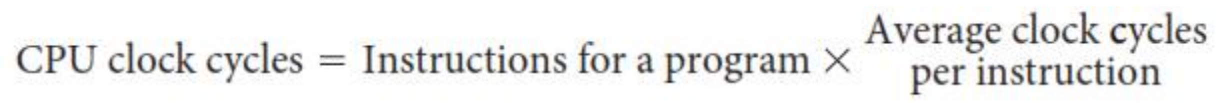
- The performance equations above did not include any reference to the number of instructions needed for the program.

- Execution time equals the number of instructions executed multiplied by the average time per instruction.

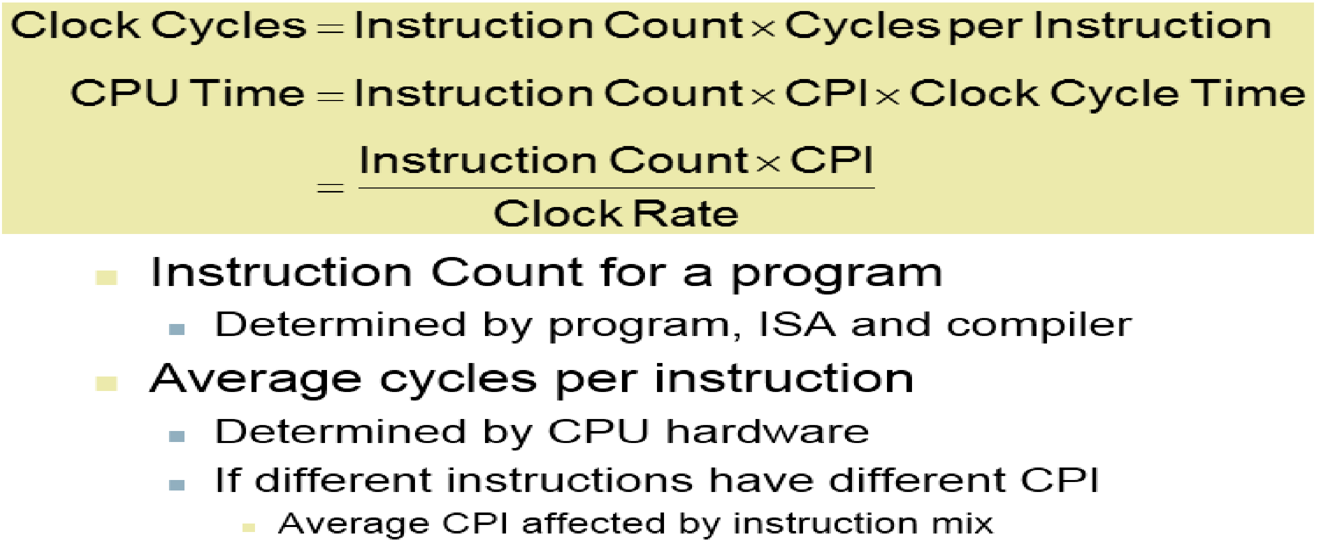
- **Clock Cycles Per Instruction (CPI):** Average number of clock cycles per instruction for a program.

- **Instruction Count (IC):**The number of instructions executed by the program.

- Therefore, the number of clock cycles required for a program can be written as



$$\text{CPU clock cycles} = \text{Instructions for a program} \times \begin{matrix}\text{Average clock cycles}\\ \text{per instruction}\end{matrix}$$



$$\begin{aligned}\text{Clock Cycles} &= \text{Instruction Count} \times \text{Cycles per Instruction}\\ \text{CPU Time} &= \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}\\ &= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}\end{aligned}$$

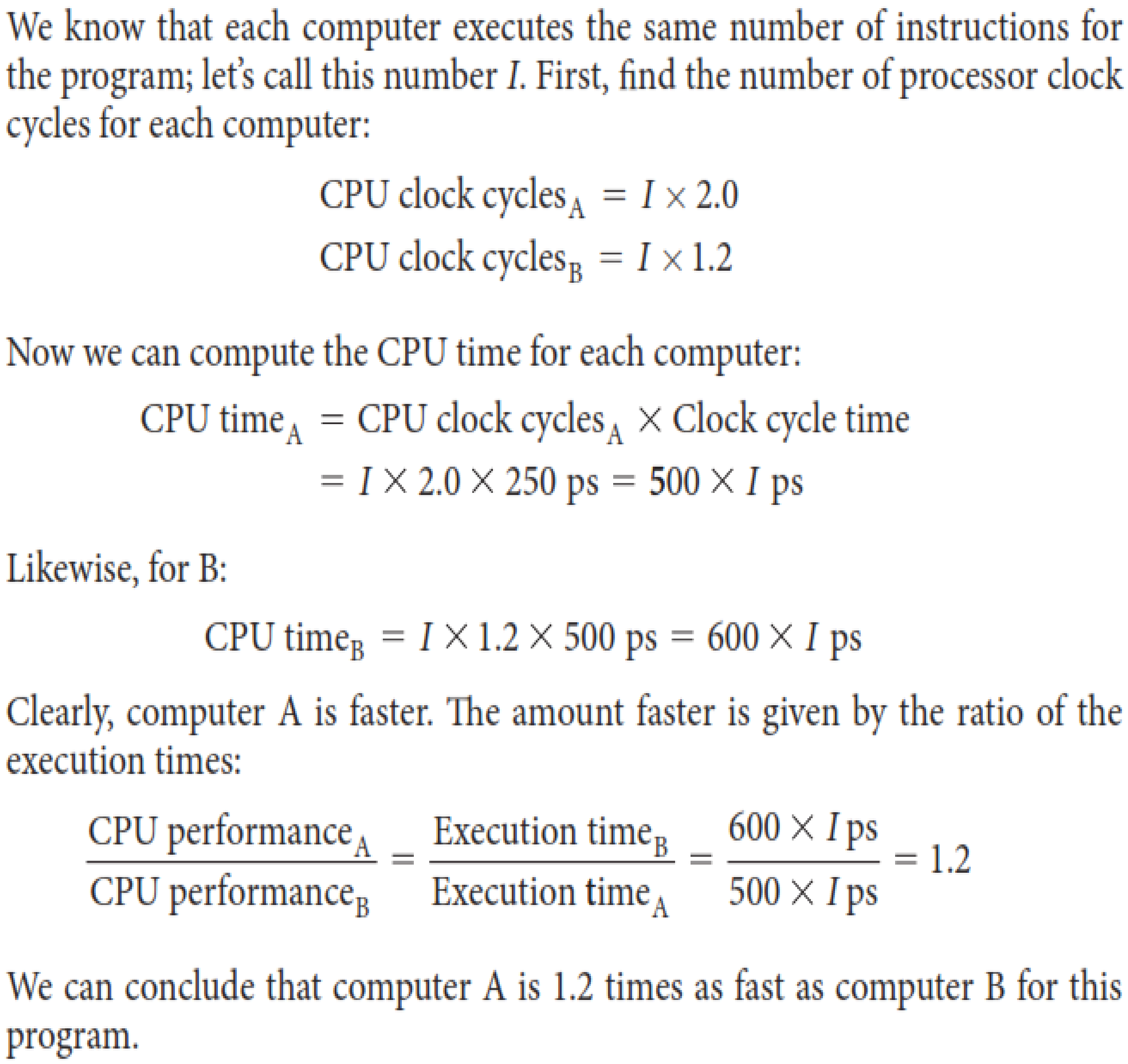
- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

Note:- *The three key factors (Instruction count , CPI, and clock cycle time) effect on the performance. We can use these formulas to compare two different implementations or to evaluate a design alternative if we know its impact on these three parameters.*

Example

Computer **A** has a **clock cycle time** of 250 ps and a **CPI** of 2.0 for some program, and computer **B** has a **clock cycle** time of 500 ps and a **CPI** of 1.2 for the same program. Which computer is faster for this program and by how much?

Answer



We know that each computer executes the same number of instructions for the program; let's call this number I. First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$
$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\begin{aligned}\text{CPU time}_A &= \text{CPU clock cycles}_A \times \text{Clock cycle time}\\ &= I \times 2.0 \times 250\text{ ps} = 500 \times I\text{ ps}\end{aligned}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500\text{ ps} = 600 \times I\text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I\text{ ps}}{500 \times I\text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

- The clock rate of frequency f is given by $f = 1/\alpha$

$$\alpha = 100ns \rightarrow f = \frac{1}{100ns} = 10MHz$$

- The size of a program is determined by its instruction count IC
- The CPI_i (cycles per instruction) represents the number of CPU cycles required by instruction i to execute.
- The average CPI of a program P:

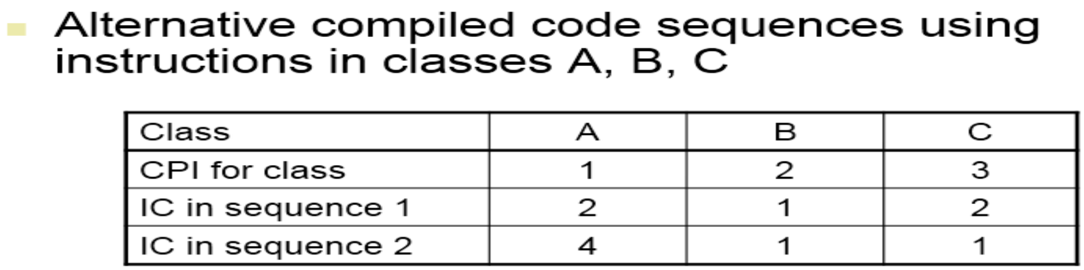
$$CPI = \sum_{i \in Classes} CPI_i \times \frac{IC_i}{IC} = \sum_{i \in Classes} CPI_i \times Freq_i$$

Ex: consider the following program:

```
ADD  A,B    CPI = 3
MUL  C,D    CPI = 4
ADD  A,C
```

- The effective CPI is $CPI = 3 \times \frac{2}{3} + 4 \times \frac{1}{3} = 3.32$

Example

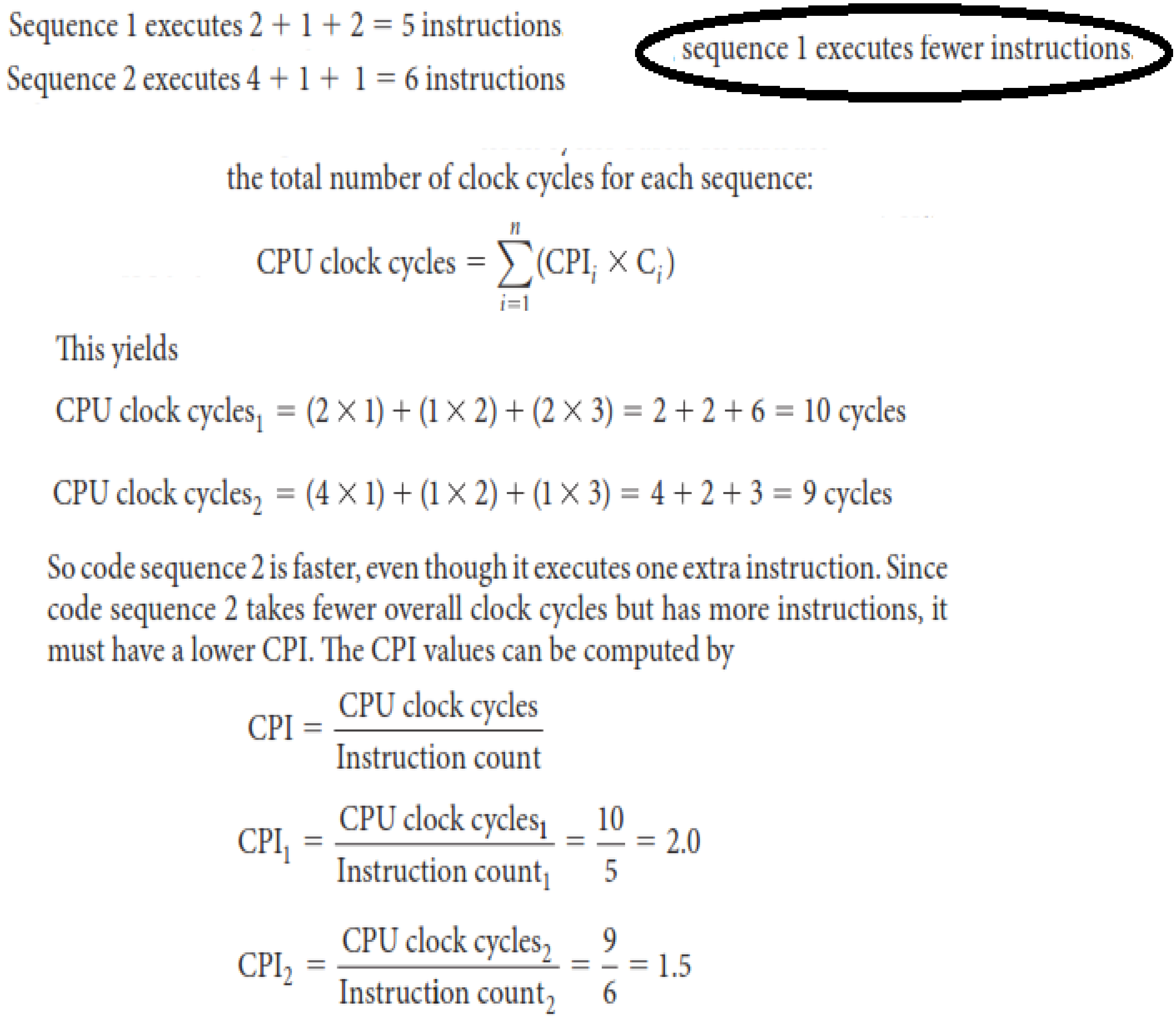


- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

Answer



Sequence 1 executes 2 + 1 + 2 = 5 instructions

Sequence 2 executes 4 + 1 + 1 = 6 instructions

sequence 1 executes fewer instructions.

the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

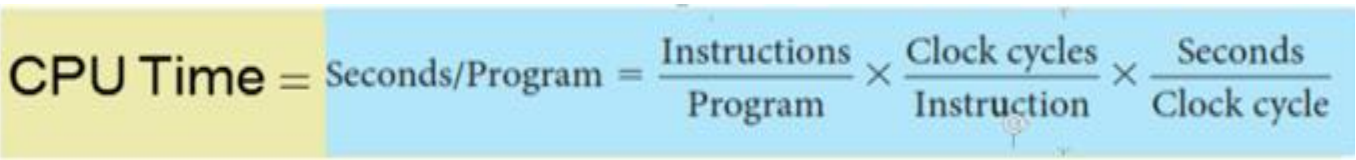
$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2.0$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

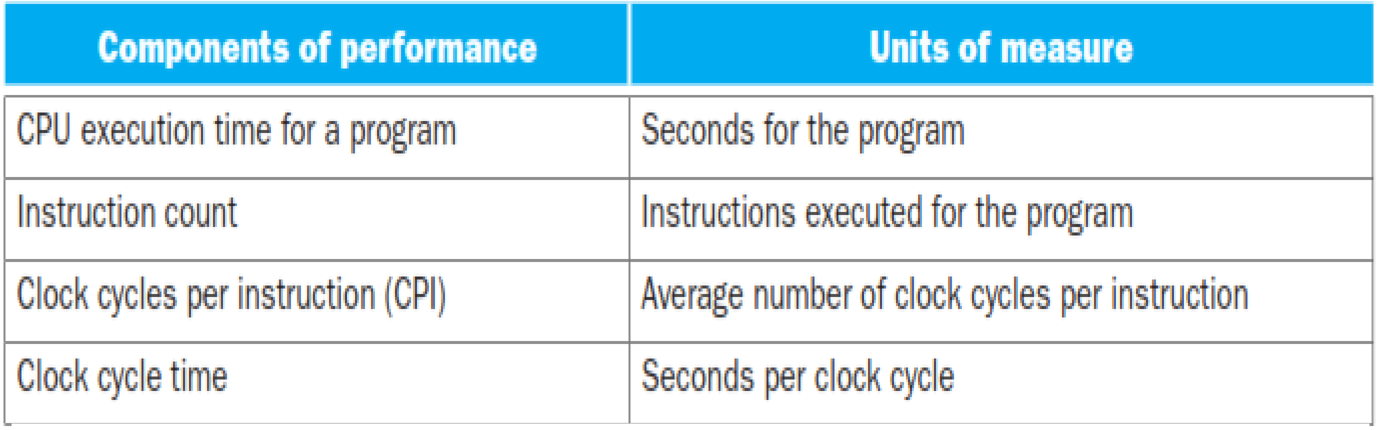


$$\text{CPU Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

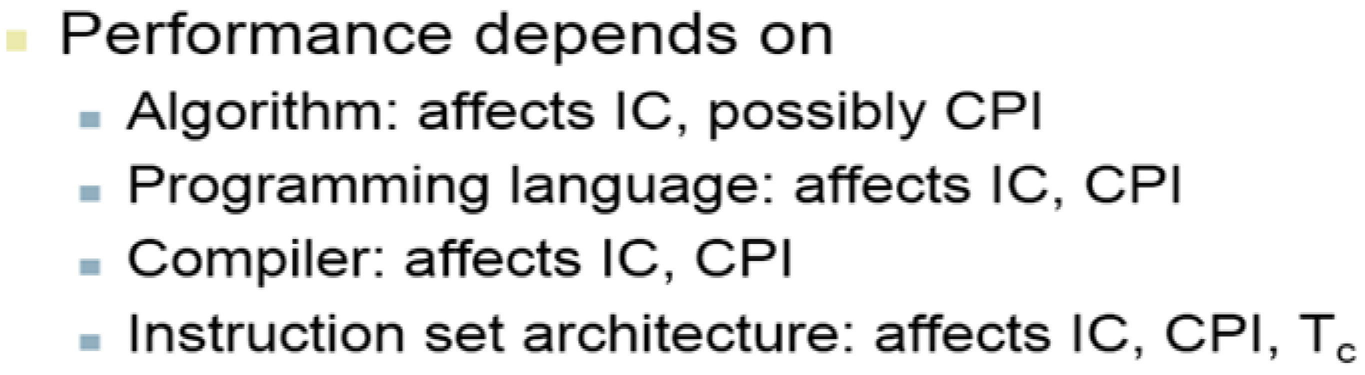
- **The Execution time depends on three factors:**
  1. How many instructions must execute to complete a program? (**Instructions per program)**
  2. How many cycles does each instruction take to execute? **Cycles per Instruction (CPI)** or reciprocal**, Insn per Cycle (IPC)**
  3. How quickly does the processor cycle? **Clock frequency** (Ghz) **(cycles per second)** or expressed as reciprocal, **Clock period** (ns) **(seconds per cycle)**

Exec. time= (Inst. /program) * (Cycles /Instr.) * (sec. /Cycle)

✓ **For minimum execution time, minimize each term**

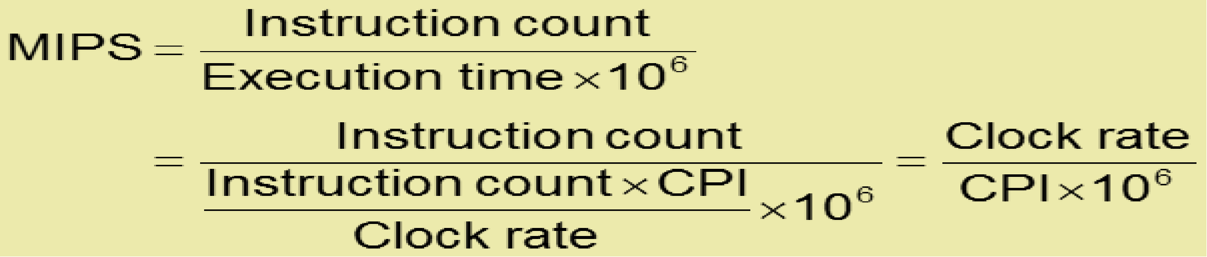


Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

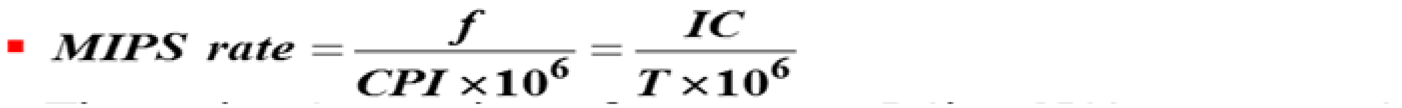


- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

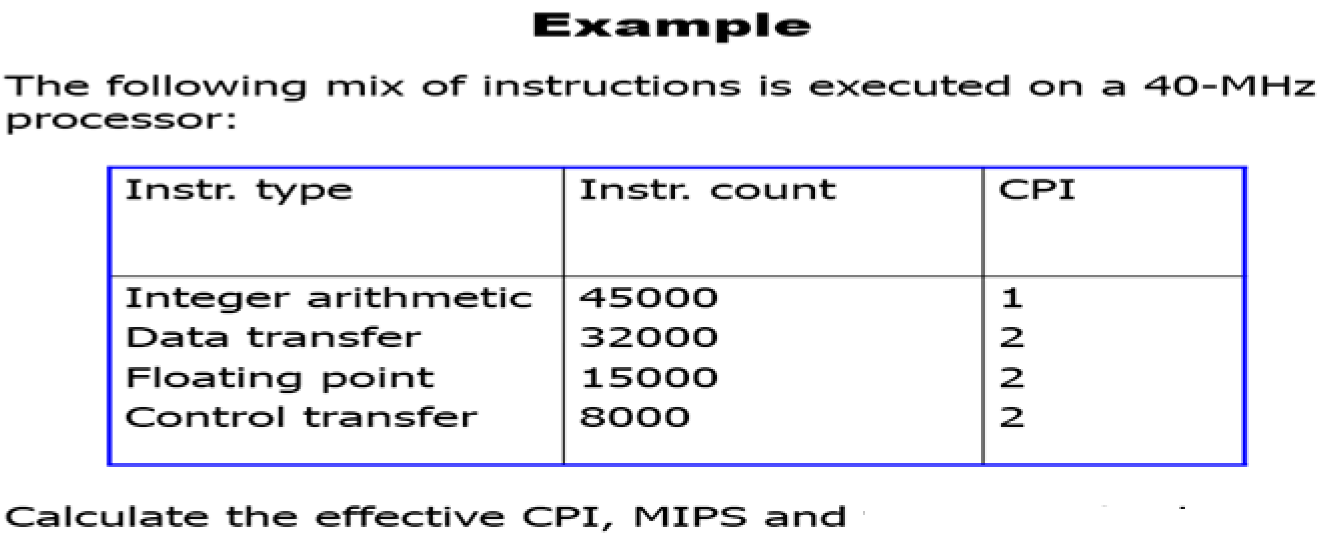
Million Instructions Per Second (MIPS): A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 10^6.



$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Instruction count}}{\dfrac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$



- $MIPS\ rate = \dfrac{f}{CPI \times 10^6} = \dfrac{IC}{T \times 10^6}$

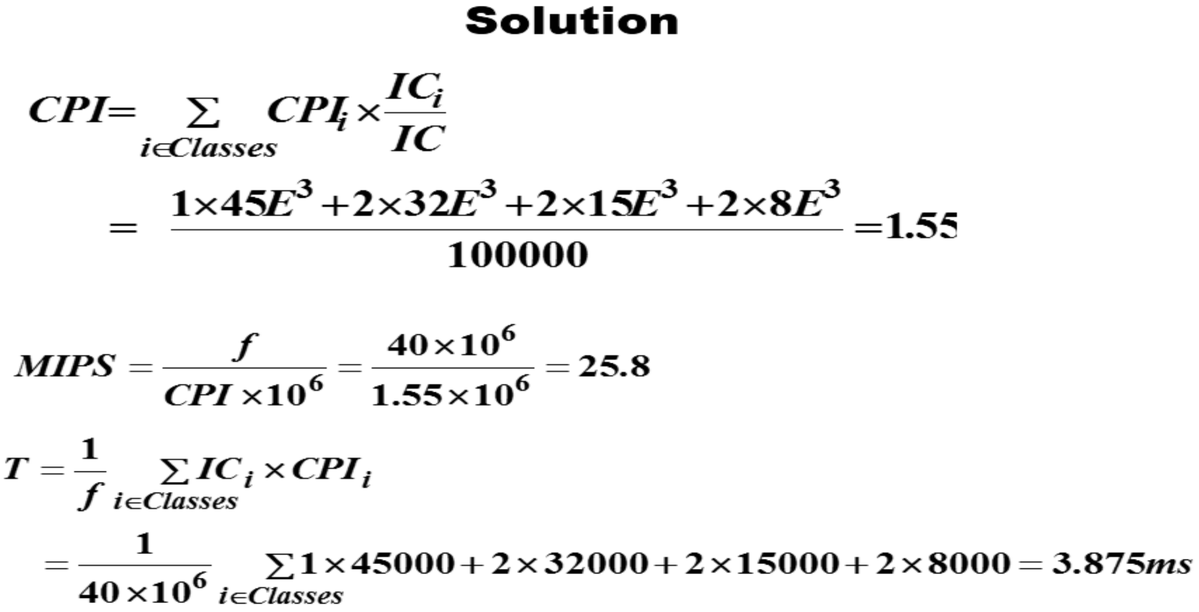


Example

The following mix of instructions is executed on a 40-MHz processor:

Instr. type	Instr. count	CPI
Integer arithmetic	45000	1
Data transfer	32000	2
Floating point	15000	2
Control transfer	8000	2

Calculate the effective CPI, MIPS and



Solution

$$CPI = \sum_{i \in Classes} CPI_i \times \frac{IC_i}{IC}$$

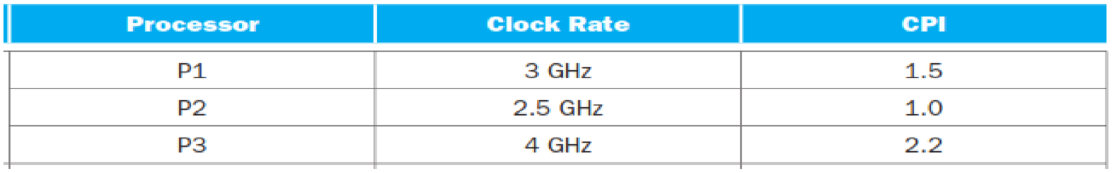
$$= \frac{1 \times 45E^3 + 2 \times 32E^3 + 2 \times 15E^3 + 2 \times 8E^3}{100000} = 1.55$$

$$MIPS = \frac{f}{CPI \times 10^6} = \frac{40 \times 10^6}{1.55 \times 10^6} = 25.8$$

$$T = \frac{1}{f} \sum_{i \in Classes} IC_i \times CPI_i$$

$$= \frac{1}{40 \times 10^6} \sum_{i \in Classes} 1 \times 45000 + 2 \times 32000 + 2 \times 15000 + 2 \times 8000 = 3.875ms$$

(Exercise 1.3): Consider three different processors P1, P2, and P3 executing the same instruction set with the clock rates and CPIs given in the following table

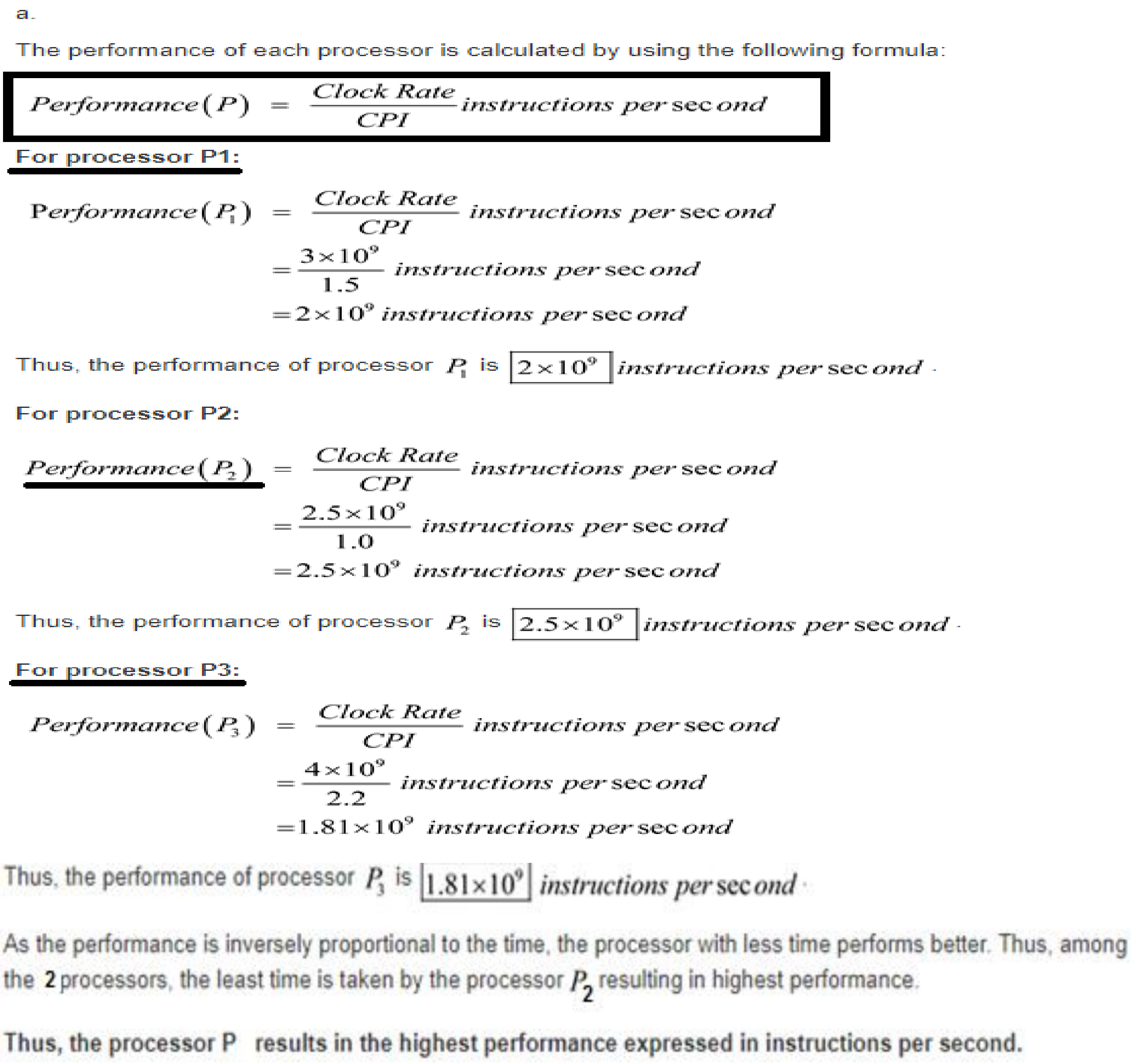


Processor	Clock Rate	CPI
P1	3 GHz	1.5
P2	2.5 GHz	1.0
P3	4 GHz	2.2

a- Which processor has the highest performance expressed in **instructions per second**?

b- If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions.

c- We are trying to reduce the time by 30% but this leads to an increase of 20% in the CPI. What clock rate should we have to get this time reduction?



a.

The performance of each processor is calculated by using the following formula:

$$\boxed{Performance(P) = \frac{Clock\ Rate}{CPI}\ instructions\ per\sec ond}$$

For processor P1:

$$\begin{aligned} Performance(P_1) &= \frac{Clock\ Rate}{CPI}\ instructions\ per\sec ond \\ &= \frac{3\times10^9}{1.5}\ instructions\ per\sec ond \\ &= 2\times10^9\ instructions\ per\sec ond \end{aligned}$$

Thus, the performance of processor P_1 is $\boxed{2\times10^9}$ $instructions\ per\sec ond$.

For processor P2:

$$\begin{aligned} Performance(P_2) &= \frac{Clock\ Rate}{CPI}\ instructions\ per\sec ond \\ &= \frac{2.5\times10^9}{1.0}\ instructions\ per\sec ond \\ &= 2.5\times10^9\ instructions\ per\sec ond \end{aligned}$$

Thus, the performance of processor P_2 is $\boxed{2.5\times10^9}$ $instructions\ per\sec ond$.

For processor P3:

$$\begin{aligned} Performance(P_3) &= \frac{Clock\ Rate}{CPI}\ instructions\ per\sec ond \\ &= \frac{4\times10^9}{2.2}\ instructions\ per\sec ond \\ &= 1.81\times10^9\ instructions\ per\sec ond \end{aligned}$$

Thus, the performance of processor P_3 is $\boxed{1.81\times10^9}$ $instructions\ per\sec ond$.

As the performance is inversely proportional to the time, the processor with less time performs better. Thus, among the 2 processors, the least time is taken by the processor P_2 resulting in highest performance.

Thus, the processor P results in the highest performance expressed in instructions per second.

b.

Consider the CPU time for executing each program is 10 seconds.

The number of cycles and number of instructions for each processor is calculated by using the following formulae:

$$\boxed{\begin{aligned} &Number\ of\ cycles(P) = Time \times Clock\ Rate \\ &Number\ of\ instructions(P) \ = \frac{Number\ of\ cycles}{CPI}\ instructions \end{aligned}}$$

For processor P1:

$$\begin{aligned} Number\ of\ cycles(P_1) &= Time \times Clock\ Rate \\ &= 10 \times 3 \times 10^9 \\ &= 30 \times 10^9 \end{aligned}$$

Thus, the number of cycles for processor P_1 is $\boxed{30 \times 10^9}$.

$$\begin{aligned} Number\ of\ instructions(P_1) &= \frac{Number\ of\ cycles}{CPI}\ instructions \\ &= \frac{30 \times 10^9}{1.5} \\ &= 20 \times 10^9 \end{aligned}$$

Thus, the number of cycles for processor P_1 is $\boxed{20 \times 10^9}$.

For processor P2:

$$\begin{aligned} Number\ of\ cycles(P_2) &= Time \times Clock\ Rate \\ &= 10 \times 2.5 \times 10^9 \\ &= 25 \times 10^9 \end{aligned}$$

Thus, the number of cycles for processor P_2 is $\boxed{25 \times 10^9}$.

$$\begin{aligned} Number\ of\ instructions(P_2) &= \frac{Number\ of\ cycles}{CPI}\ instructions \\ &= \frac{25 \times 10^9}{1.0} \\ &= 25 \times 10^9 \end{aligned}$$

Thus, the number of instructions for processor P_2 is $\boxed{25 \times 10^9}$.

For processor P3:

$$\begin{aligned} Number\ of\ cycles(P_3) &= Time \times Clock\ Rate \\ &= 10 \times 4 \times 10^9 \\ &= 40 \times 10^9 \end{aligned}$$

Thus, the number of cycles for processor P_3 is $\boxed{40 \times 10^9}$.

$$\begin{aligned} Number\ of\ instructions(P_3) &= \frac{Number\ of\ cycles}{CPI}\ instructions \\ &= \frac{40 \times 10^9}{2.2} \\ &= 18.18 \times 10^9 \end{aligned}$$

Thus, the number of instructions for processor P_3 is $\boxed{18.18 \times 10^9}$.

For processor P1:

$$\begin{aligned} CPI &= 1.2 \times CPI \\ &= 1.2 \times 1.5 \\ &= 1.8 \end{aligned}$$

$$CPI = \boxed{1.8}$$

$$\begin{aligned} Number\ of\ cycles(P_1) &= Time \times Clock\ Rate \\ &= 10 \times 3 \times 10^9 \\ &= 30 \times 10^9 \end{aligned}$$

Thus, the number of cycles for processor P_1 is $\boxed{30\times10^9}$.

$$\begin{aligned} Number\ of\ instructions(P_1) &= \frac{Number\ of\ cycles}{CPI}\ instructions \\ &= \frac{30\times10^9}{1.5} \\ &= 20\times10^9 \end{aligned}$$

Thus, the number of instructions for processor P_1 is $\boxed{20\times10^9}$.

$$\begin{aligned} Clock\ rate(P1) &= \frac{(Number\ of\ instructions \times CPI)}{Time} \\ &= \frac{(20\times10^9\times1.8)}{7} \\ &= \frac{36000000000}{7} \\ &= 5.14\,GHz \end{aligned}$$

Thus, the Clock rate for processor P1 is $\boxed{5.14\,\mathrm{GHz}}$.

c.

Consider the old CPU time is 10 seconds.

Now, calculate the new CPU time as follows:

$$\boxed{CPU\ Time = \frac{(I \times CPI)}{clock\ rate}}$$

The time is decreased by 30%.

$$\begin{aligned} t_1 &= \frac{70\times t}{100} \\ &= 0.7t \end{aligned}$$

So, the CPU time is 7s.

CPI is increased by 20%.

$$\begin{aligned} CPI &= \frac{(120\times CPI)}{100} \\ &= 1.2 \times CPI \end{aligned}$$

So, $CPI = 1.2\times CPI$.

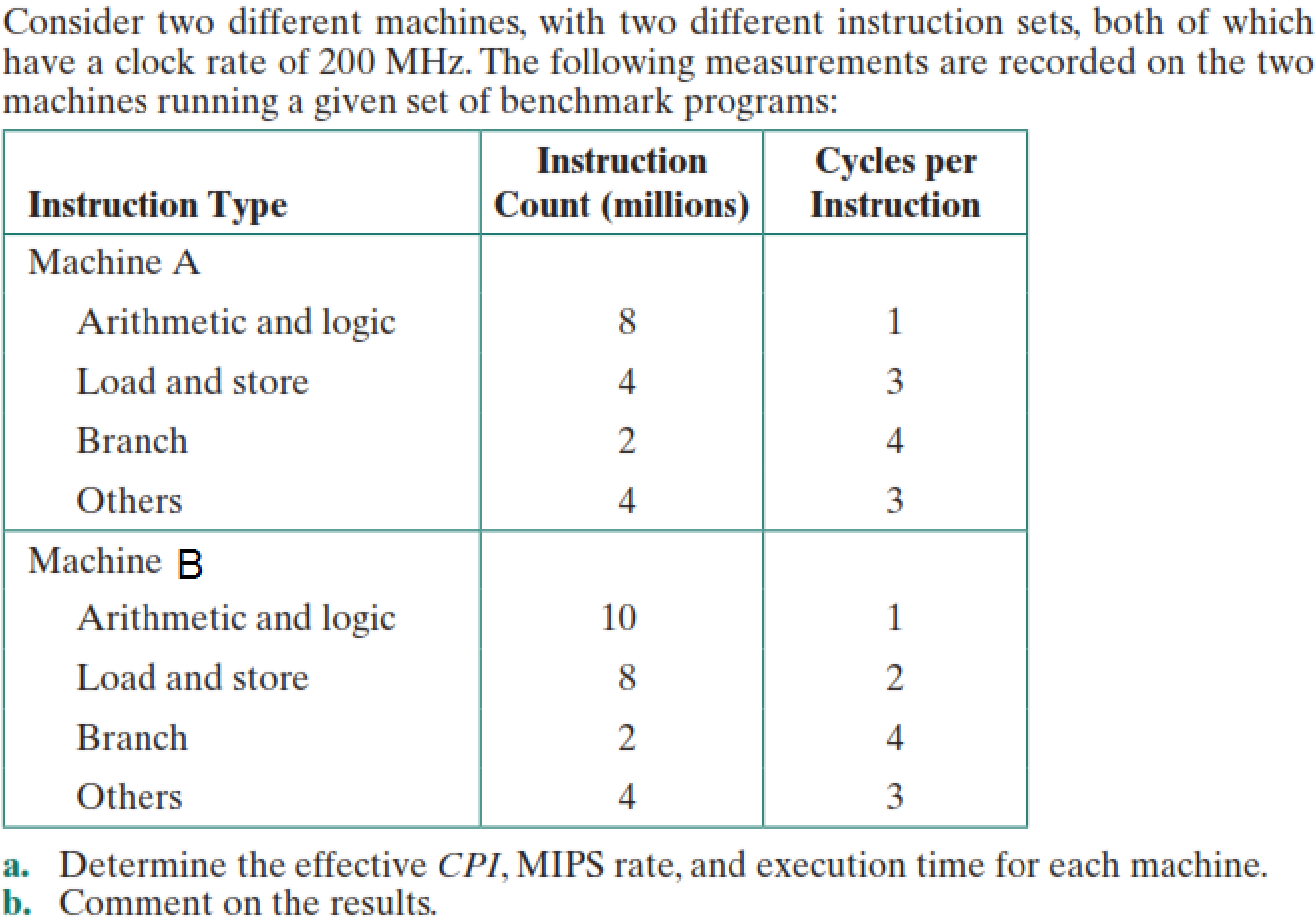
Calculate the clock rate to get the time reduction by using the following formula:

$$\boxed{Clock\ rate = \frac{(Number\ of\ instruction \times CPI)}{Time}}$$

Calculate number of cycles and number 0f instructions of each processor by using the following formulae:

$$\boxed{\begin{aligned} &Number\ of\ cycles(P) = Time \times Clock\ Rate \\ &Number\ of\ instructions(P) = \frac{Number\ of\ cycles}{CPI}\ instructions \end{aligned}}$$

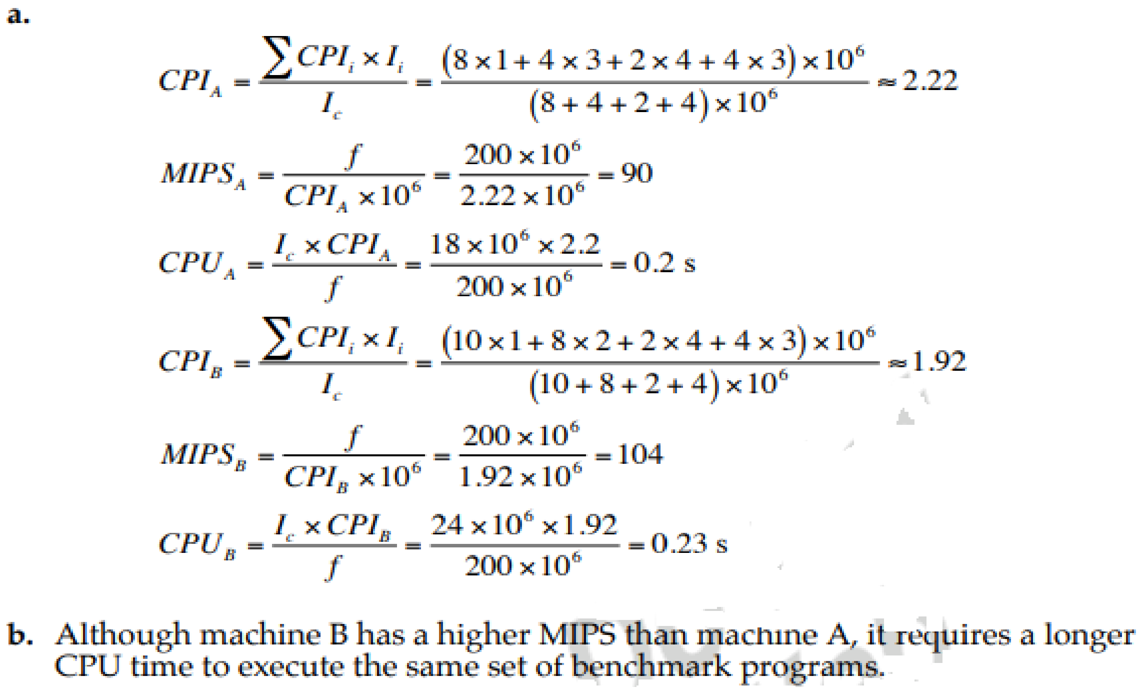
Example



Consider two different machines, with two different instruction sets, both of which have a clock rate of 200 MHz. The following measurements are recorded on the two machines running a given set of benchmark programs:

Instruction Type	Instruction Count (millions)	Cycles per Instruction
Machine A		
Arithmetic and logic	8	1
Load and store	4	3
Branch	2	4
Others	4	3
Machine B		
Arithmetic and logic	10	1
Load and store	8	2
Branch	2	4
Others	4	3

a. Determine the effective *CPI*, MIPS rate, and execution time for each machine.
b. Comment on the results.



a.

$$CPI_A = \frac{\sum CPI_i \times I_i}{I_c} = \frac{(8\times1+4\times3+2\times4+4\times3)\times10^6}{(8+4+2+4)\times10^6} \approx 2.22$$

$$MIPS_A = \frac{f}{CPI_A \times 10^6} = \frac{200\times10^6}{2.22\times10^6} = 90$$

$$CPU_A = \frac{I_c \times CPI_A}{f} = \frac{18\times10^6\times2.2}{200\times10^6} = 0.2\text{ s}$$

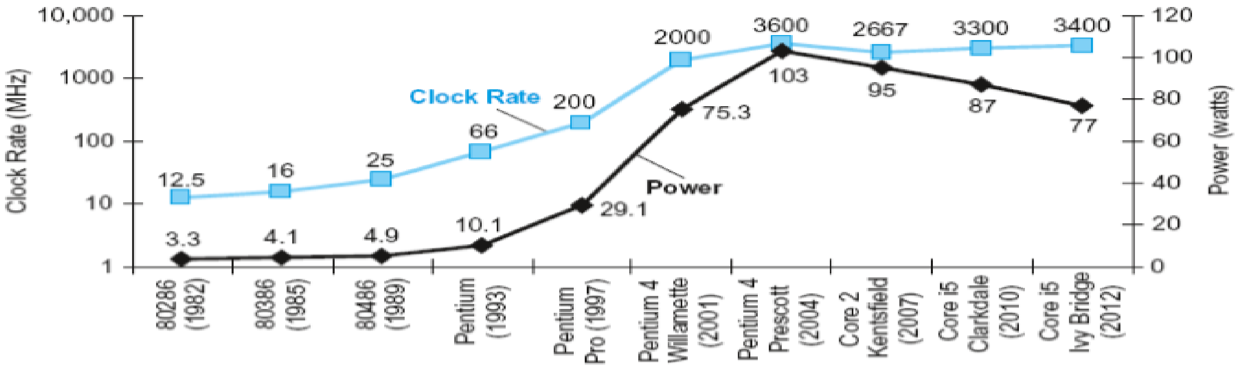
$$CPI_B = \frac{\sum CPI_i \times I_i}{I_c} = \frac{(10\times1+8\times2+2\times4+4\times3)\times10^6}{(10+8+2+4)\times10^6} \approx 1.92$$

$$MIPS_B = \frac{f}{CPI_B \times 10^6} = \frac{200\times10^6}{1.92\times10^6} = 104$$

$$CPU_B = \frac{I_c \times CPI_B}{f} = \frac{24\times10^6\times1.92}{200\times10^6} = 0.23\text{ s}$$

b. Although machine B has a higher MIPS than machine A, it requires a longer CPU time to execute the same set of benchmark programs.

Power Trends



- In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

- For CMOS chips, traditional dominant energy consumption has been in switching transistors, called *dynamic power*

- Because leakage current flows even when a transistor is off, now ***static power*** important too

$$Power_{static} = Current_{static} \times Voltage$$

- Leakage current increases in processors with smaller transistor sizes
- Increasing the number of transistors increases power even if they are turned off

Reducing Power

- Suppose a new CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

- The power wall
  - We can't reduce voltage further
  - We can't remove more heat
- How else can we improve performance?

Exercise 1.8

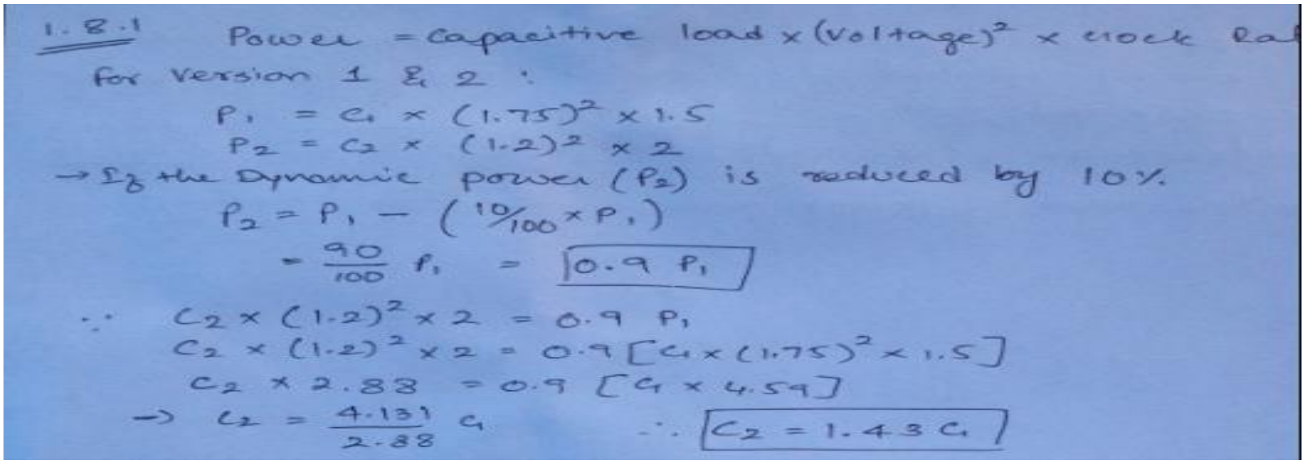
Suppose we have developed new versions of a processor with the following characteristics.

	Version	Voltage	Clock Rate
a.	Version 1	1.75 V	1.5 GHz
	Version 2	1.2 V	2 GHz

1.8.1 How much has the capacitive load varied between versions if the dynamic power has been reduced by 10%?

1.8.2 How much has the dynamic power been reduced if the capacitive load does not change?

1.8.3 Assuming that the capacitive load of version 2 is 80% the capacitive load of version 1, find the voltage for version 2 if the dynamic power of version 2 is reduced by 40% from version 1.



1.8.1 Power = Capacitive load × $(\text{Voltage})^2$ × clock Ra

for version 1 & 2 :

$P_1 = C_1 \times (1.75)^2 \times 1.5$

$P_2 = C_2 \times (1.2)^2 \times 2$

→ If the Dynamic power (P_2) is reduced by 10%.

$P_2 = P_1 - (10/100 \times P_1)$

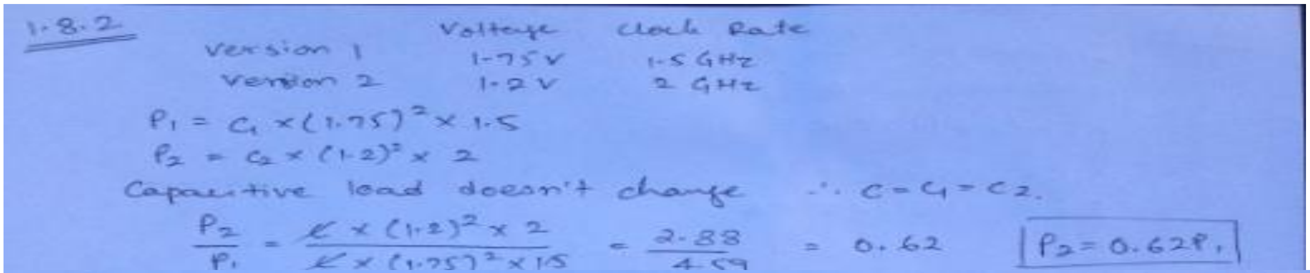
$= \frac{90}{100} P_1 = \boxed{0.9\ P_1}$

$\because C_2 \times (1.2)^2 \times 2 = 0.9\ P_1$

$C_2 \times (1.2)^2 \times 2 = 0.9[C_1 \times (1.75)^2 \times 1.5]$

$C_2 \times 2.88 = 0.9\ [C_1 \times 4.59]$

$\rightarrow C_2 = \frac{4.131}{2.88} C_1 \quad \therefore \boxed{C_2 = 1.43\ C_1}$



1.8.2

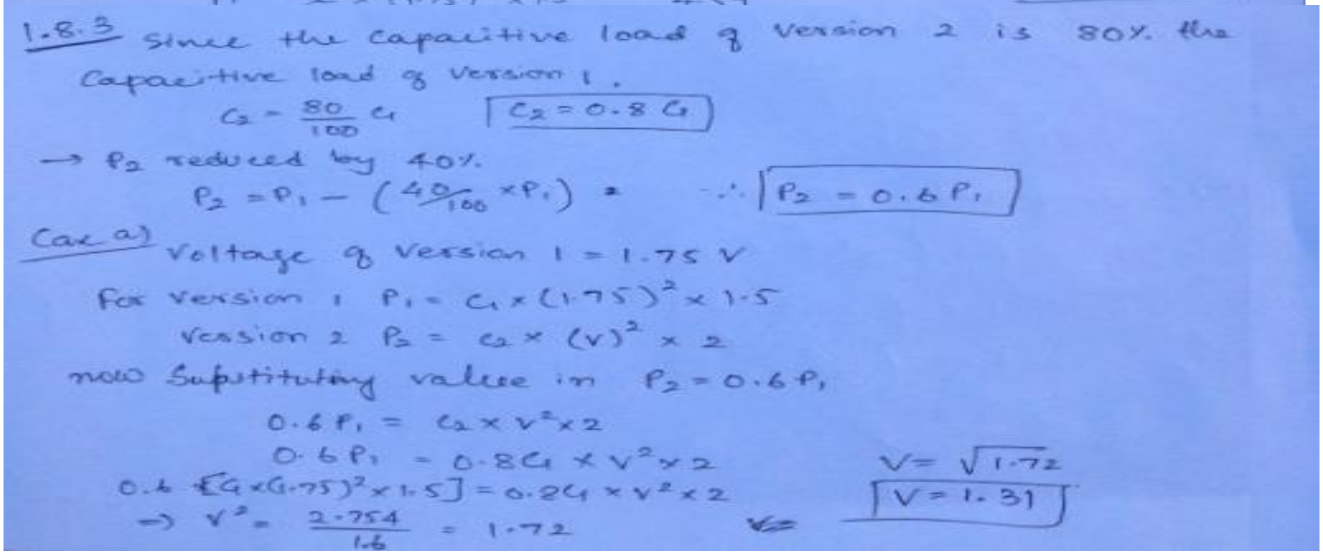
	Voltage	Clock Rate
Version 1	1.75 V	1.5 GHz
Version 2	1.2 V	2 GHz

$P_1 = C_1 \times (1.75)^2 \times 1.5$

$P_2 = C_2 \times (1.2)^2 \times 2$

Capacitive load doesn't change $\therefore C = C_1 = C_2$.

$\frac{P_2}{P_1} = \frac{\cancel{C} \times (1.2)^2 \times 2}{\cancel{C} \times (1.75)^2 \times 1.5} = \frac{2.88}{4.59} = 0.62 \quad \boxed{P_2 = 0.62 P_1}$



1.8.3 Since the capacitive load of Version 2 is 80% the Capacitive load of Version 1.

$C_2 = \frac{80}{100} C_1 \quad \boxed{C_2 = 0.8\ C_1}$

→ P_2 reduced by 40%.

$P_2 = P_1 - (40/100 \times P_1) \quad \therefore \boxed{P_2 = 0.6 P_1}$

Case a) Voltage of Version 1 = 1.75 V

For version 1 $P_1 = C_1 \times (1.75)^2 \times 1.5$

Version 2 $P_2 = C_2 \times (V)^2 \times 2$

now Substituting value in $P_2 = 0.6 P_1$

$0.6 P_1 = C_2 \times V^2 \times 2$

$0.6 P_1 = 0.8 C_1 \times V^2 \times 2$

$0.6 [C_1 \times (1.75)^2 \times 1.5] = 0.8 C_1 \times V^2 \times 2$

$\rightarrow V^2 = \frac{2.754}{1.6} = 1.72$

$V = \sqrt{1.72}$

$\boxed{V = 1.31}$

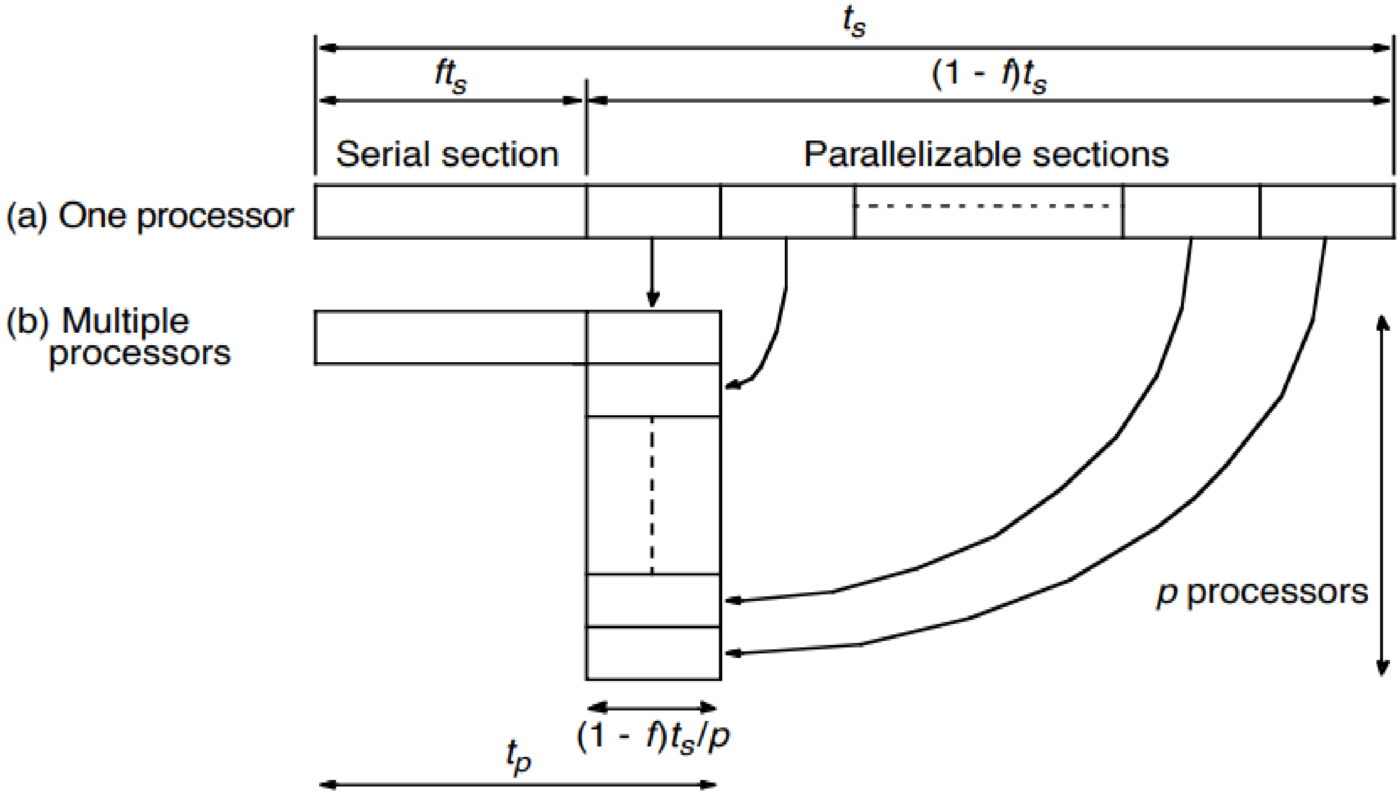
Amdahl’s Law

While manufacturers make enormous efforts on improving the performance of processors, input/output devices like memory and storage devices are still too slow compared to the processors. This means that the overall speed improvement is limited by the low speed of input/output devices. In other words, at a certain point, manufacturers will need to pay more attention in improving I/O speed instead of processor's speed.

What is Amdahl’s Law?

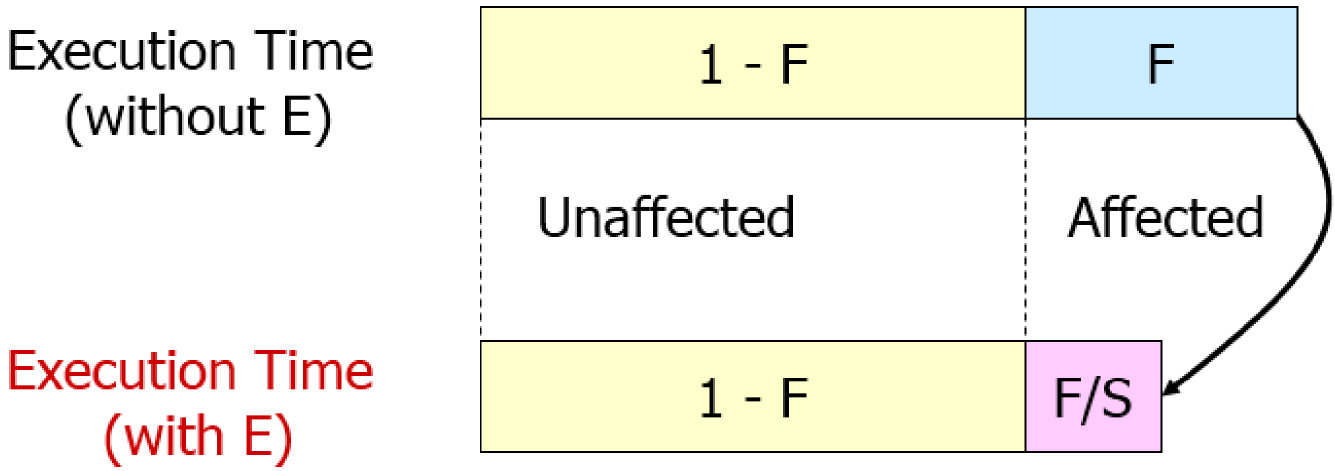
Amdahl's law is an expression used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors.

Bounds on Speedup



Single Enhancement

F: Fraction enhanced, S: Speedup enhanced



$$Speedup = \frac{1}{(1-F)+\frac{F}{S}}$$

- **<u>Speedup</u>**: is the maximum possible improvement of the system.
- **<u>F</u> (Fraction)**: is the part that can be improved. In other words, (1-F) is the part of the system that cannot be improved.
- **<u>S</u> (factor of improvement)**: is the performance improvement factor of **<u>F</u>** after applying the enhancements.

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

- Relative to how it performed previously

$$\boxed{\text{Speedup(E)} = \frac{\text{Performance with E}}{\text{Performance before}} = \frac{\text{ExTime before}}{\text{ExTime with E}}}$$

- Enhancement improves a fraction *f* of execution time by a factor *s* *(speedup E(f))* and the remaining time is unaffected

$$\boxed{\text{ExTime with E} = \text{ExTime before} \times (f\,/\,s + (1 - f\,))}$$

$$\boxed{\text{Speedup(E)} = \frac{1}{(f\,/\,s + (1 - f\,))}}$$

Computer Performance

"X is N% faster than Y."

$$\frac{\text{Execution Time of Y}}{\text{Execution Time of X}} = 1+\frac{N}{100}$$

Using Amdahl's law

Overall speedup if we make 90% of a program run 10 times faster.

$F = 0.9 \quad S = 10$

$$\text{Overall Speedup} = \frac{1}{(1-0.9)+\frac{0.9}{10}} = \frac{1}{0.1+0.09} = 5.26$$

Overall speedup if we make 80% of a program run 20% faster.

$F = 0.8 \quad S = 1.2$

$$\text{Overall Speedup} = \frac{1}{(1-0.8)+\frac{0.8}{1.2}} = \frac{1}{0.2+0.66} = 1.153$$

Example

Let a program have 40 percent of its code enhanced (so F = 0.4) to run 2.3 times faster (so S= 2.3). What is the overall system speedup E?

Solution

Step 1: Setup the equation: $\mathbf{E} = ((1 - \mathbf{F}) + (\mathbf{F} / \mathbf{S}))^{-1}$

Step 2: Plug in values & solve $\mathbf{E} = ((1 - 0.4) + (0.4 / 2.3))^{-1}$

$$= (0.6 + 0.174)^{-1} = 1 / 0.774$$

$$= \mathbf{1.292}$$

Example

Suppose a program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of this time.

a- How much do I have to improve the speed of multiplication if I want my program to run four times faster?

b- How much do I have to improve the speed of multiplication if I want my program to run five times faster?

Answer

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

$$\frac{\mathbf{100}}{\mathbf{4}} = (\mathbf{100} - \mathbf{80}) + \frac{\mathbf{80}}{\boldsymbol{s}}$$

$$\mathbf{25 = 20} + \frac{\mathbf{80}}{\boldsymbol{s}}$$

$\mathbf{5} = \frac{\mathbf{80}}{\boldsymbol{s}}$ → **s= 16**

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

For this problem:

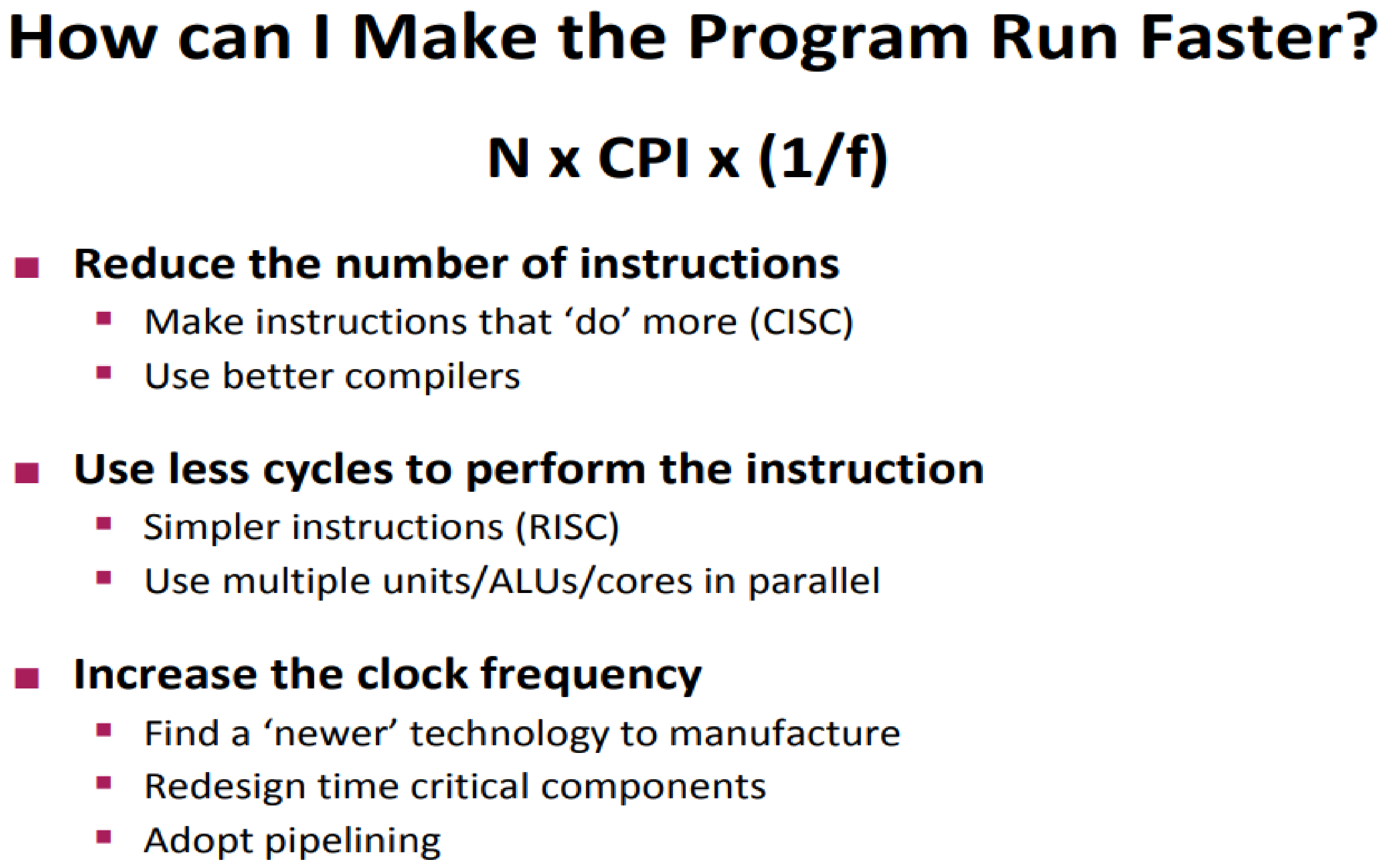
$$\text{Execution time after improvement} = \frac{80 \text{ seconds}}{n} + (100 - 80 \text{ seconds})$$

Since we want the performance to be five times faster, the new execution time should be 20 seconds, giving

$$20 \text{ seconds} = \frac{80 \text{ seconds}}{n} + 20 \text{ seconds}$$

$$0 = \frac{80 \text{ seconds}}{n}$$

That is, there is *no amount* by which we can enhance-multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload.



How can I Make the Program Run Faster?

N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel
- **Increase the clock frequency**
  - Find a ‘newer’ technology to manufacture
  - Redesign time critical components
  - Adopt pipelining

Homework # 1

- **Exercises in the Textbook (Computer Organization & Design, by Patterson & Hennessy, 5th Edition).**
- **1.3**
- **1.5**
- **1.6**
- **1.7**
- **1.8**
- **1.10**
- **1.14**

Chapter 2
Instructions (Language of the Computer)

- The words of a computer’s language are called **instructions**, and its vocabulary is called an ***instruction set***.

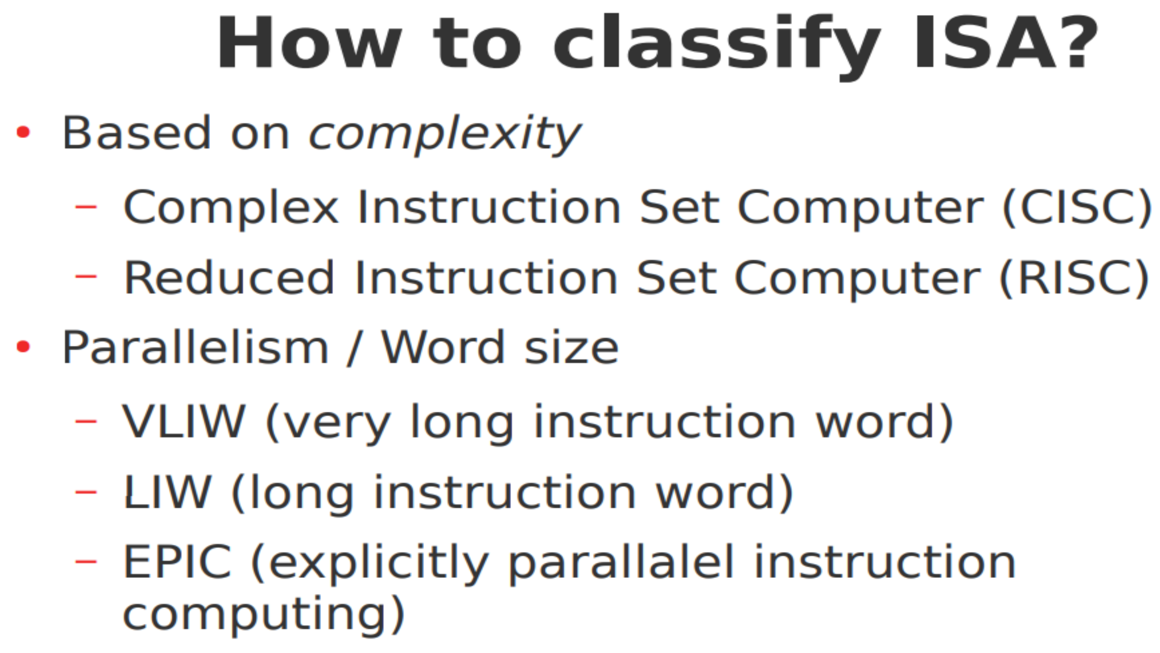
- The instruction set, also called ISA (*Instruction Set Architecture*), is part of a computer that pertains to programming, which is basically machine language.

- The instruction set provides commands to the processor, to tell it what it needs to do.

- The instruction set consists of addressing modes, instructions data types, registers, memory architecture, interrupt, and exception handling, and external I/O.

- An instruction set can be built into the hardware of the processor, or it can be emulated in software, using an interpreter. The hardware design is more efficient and faster for running programs than the emulated software.

- Computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and energy.



How to classify ISA?

- Based on *complexity*
  - Complex Instruction Set Computer (CISC)
  - Reduced Instruction Set Computer (RISC)
- Parallelism / Word size
  - VLIW (very long instruction word)
  - LIW (long instruction word)
  - EPIC (explicitly parallalel instruction computing)

RISC vs CISC

❑ **RISC (reduced instruction set computer) instructions**

- ✓ only load/store instructions access memory
- ✓ operands (data) must be in registers to perform operation
- ✓ each instruction roughly taking same amount of time
- ✓ simple addressing modes

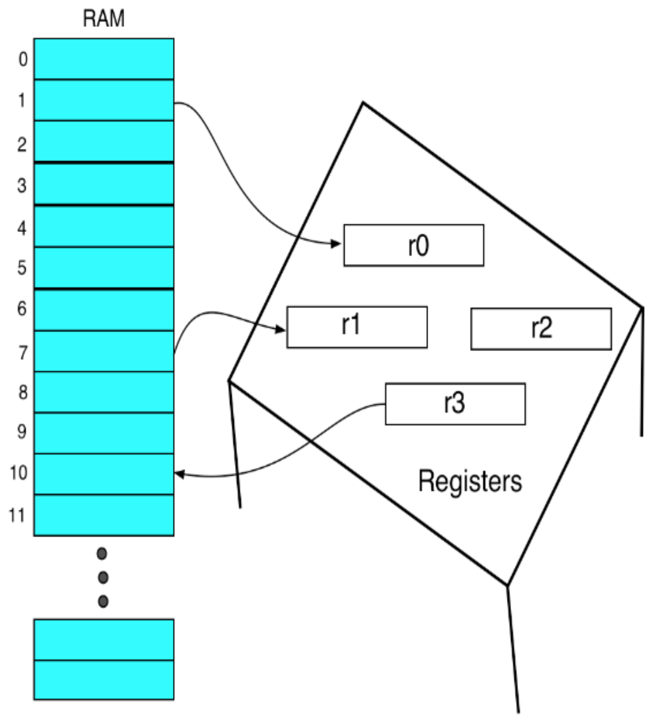
❑ **CISC (complex instruction set computer) instructions**

- ✓ ALU instructions access memory to fetch operands
- ✓ load/store instructions access memory
- ✓ some instructions' execution time is much longer than other instructions
- ✓ complex addressing modes

Registers vs. memory

Data can be stored in **registers or memory locations**. Memory access is slower (takes approximately 50 ns) than register access (takes approximately 1 ns or less).

To increase the speed of computation it pays to keep the variables in registers as long as possible. However, due to technology limitations, the number of registers is quite limited (typically 8-64).



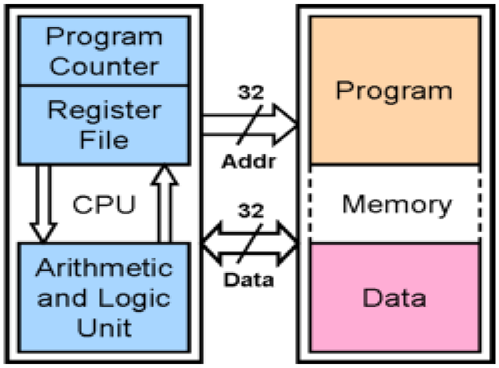
Memory can be viewed as a bookshelf

View registers as spaces on your table

MIPS registers

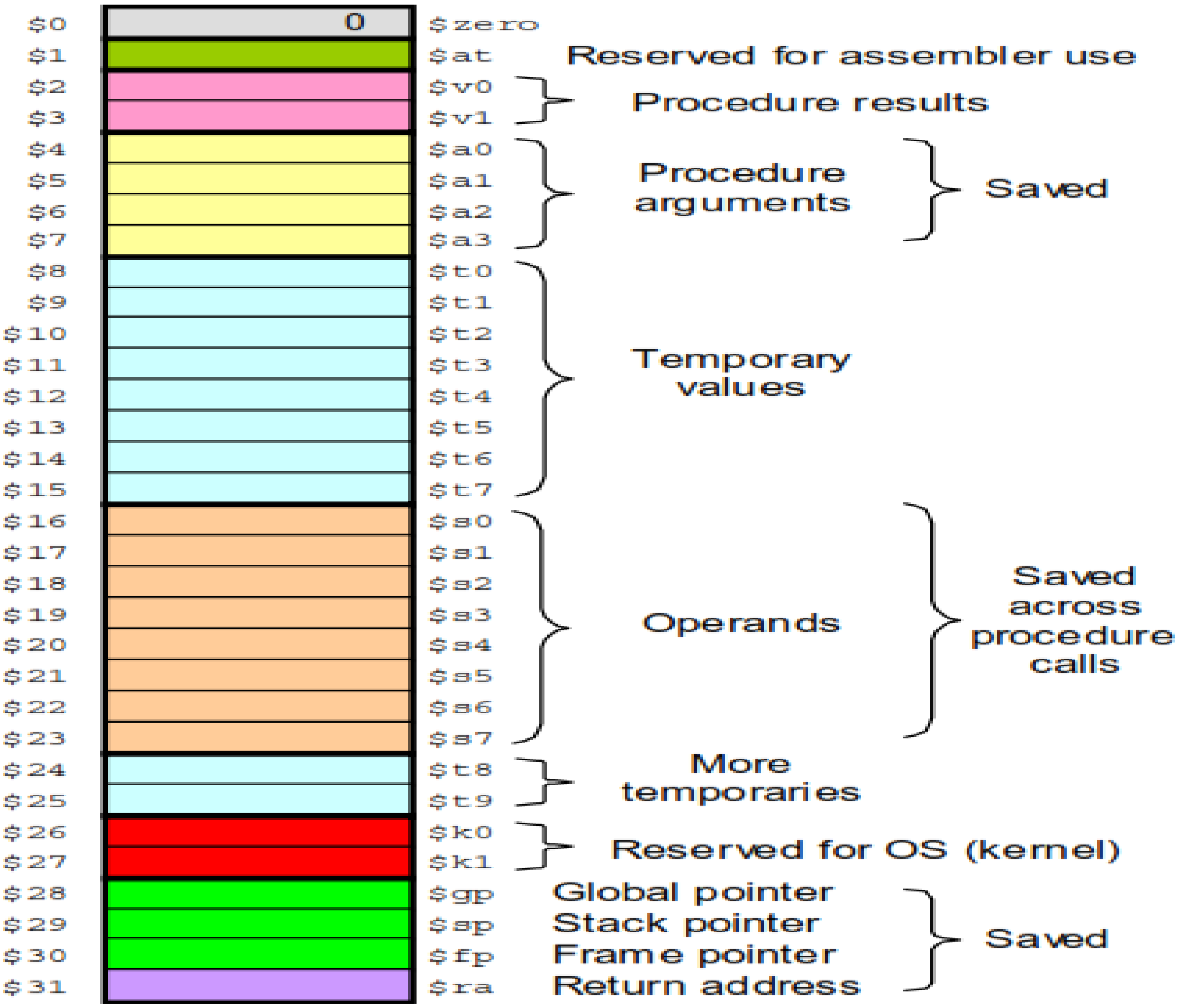
MIPS has 32 registers r0-r31.

Memory and Registers

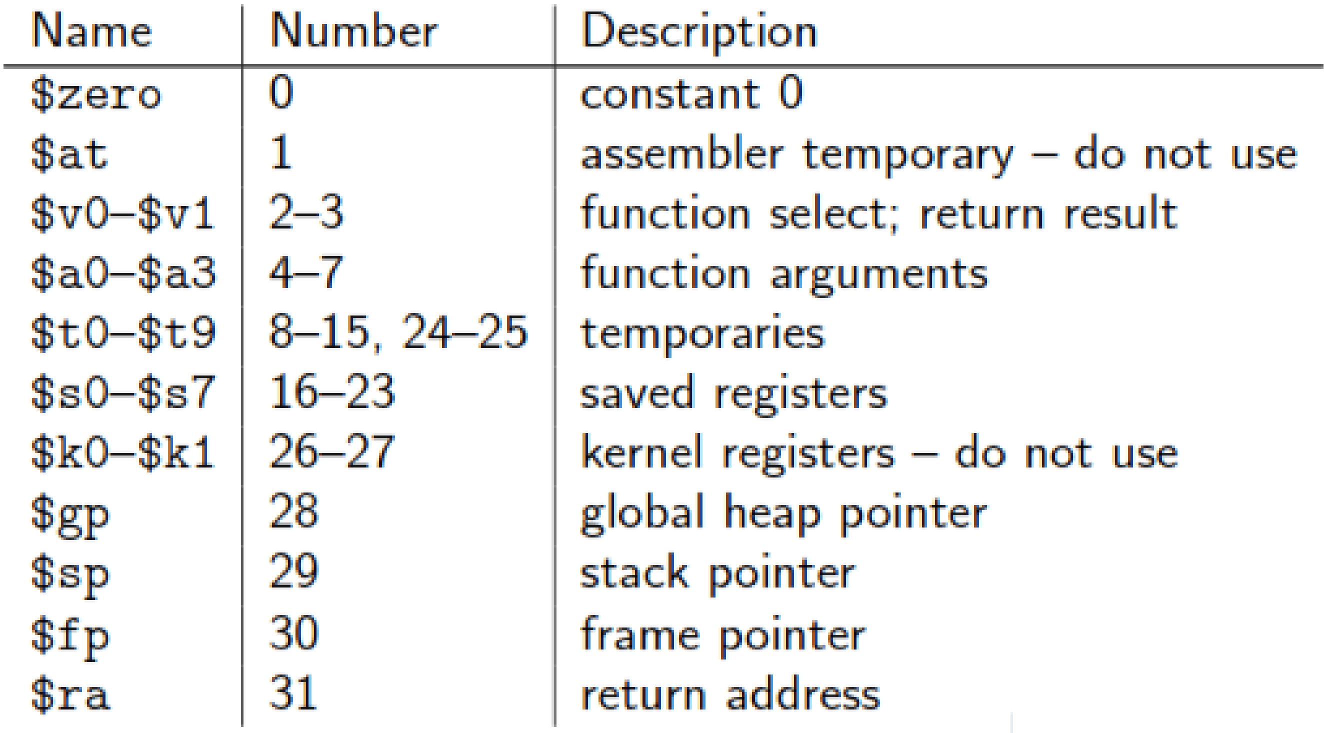


- Addresses are 32-bits words
  – 2^{32} different locations
- Words are 32-bits, or 4 bytes
  – 2^{30} addressable words
  - Word address must be aligned
- Registers are also word-sized
  - Only 32 general purpose
  - Special: PC, Status, HI, LO, . . .
  - Floating point registers, . . .

MIPS registers

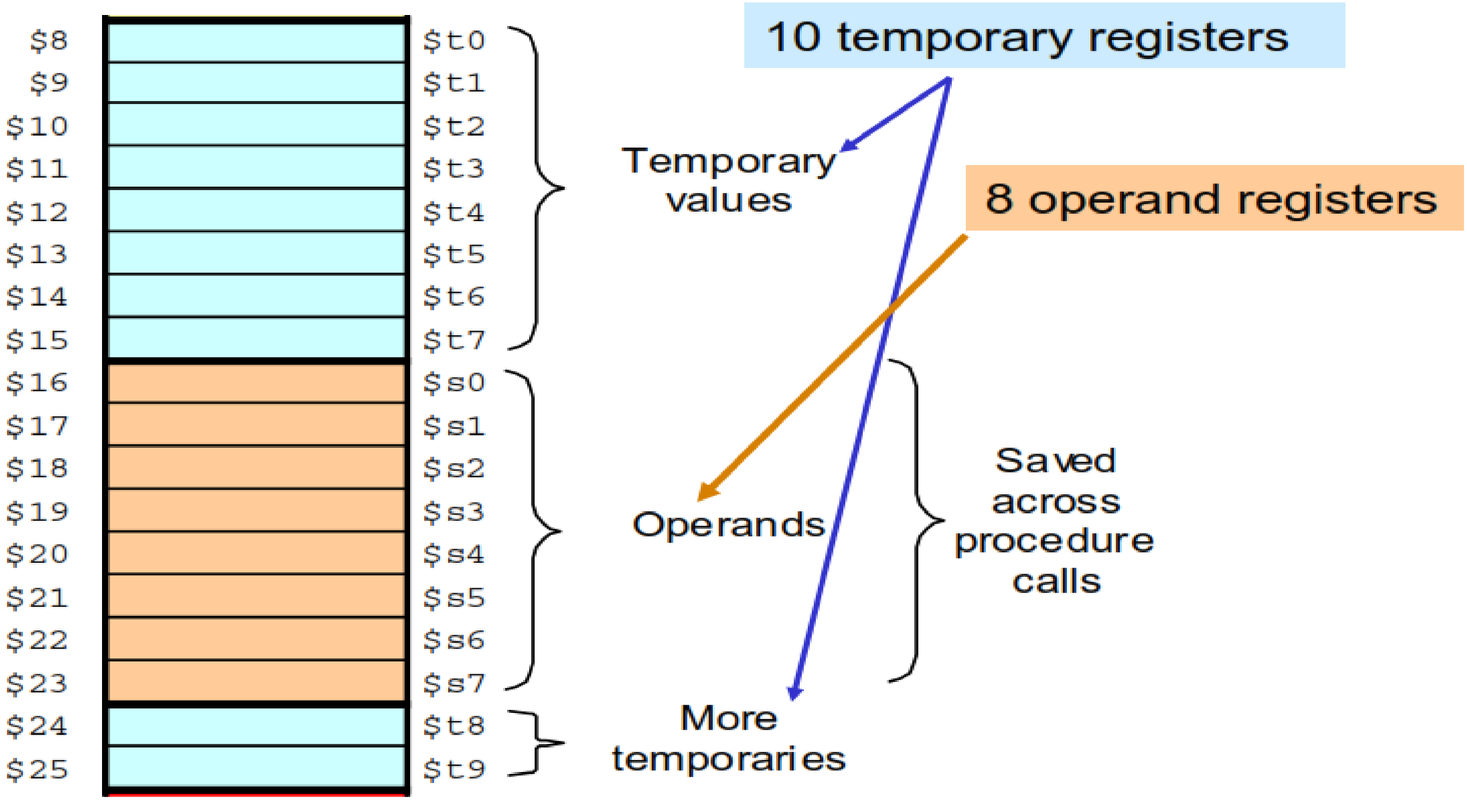


$0	0	$zero		
$1		$at	Reserved for assembler use	
$2		$v0	Procedure results	
$3		$v1		
$4		$a0	Procedure arguments	Saved
$5		$a1		
$6		$a2		
$7		$a3		
$8		$t0	Temporary values	
$9		$t1		
$10		$t2		
$11		$t3		
$12		$t4		
$13		$t5		
$14		$t6		
$15		$t7		
$16		$s0	Operands	Saved across procedure calls
$17		$s1		
$18		$s2		
$19		$s3		
$20		$s4		
$21		$s5		
$22		$s6		
$23		$s7		
$24		$t8	More temporaries	
$25		$t9		
$26		$k0	Reserved for OS (kernel)	
$27		$k1		
$28		$gp	Global pointer	Saved
$29		$sp	Stack pointer	
$30		$fp	Frame pointer	
$31		$ra	Return address	



Name	Number	Description
$zero	0	constant 0
$at	1	assembler temporary – do not use
$v0–$v1	2–3	function select; return result
$a0–$a3	4–7	function arguments
$t0–$t9	8–15, 24–25	temporaries
$s0–$s7	16–23	saved registers
$k0–$k1	26–27	kernel registers – do not use
$gp	28	global heap pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

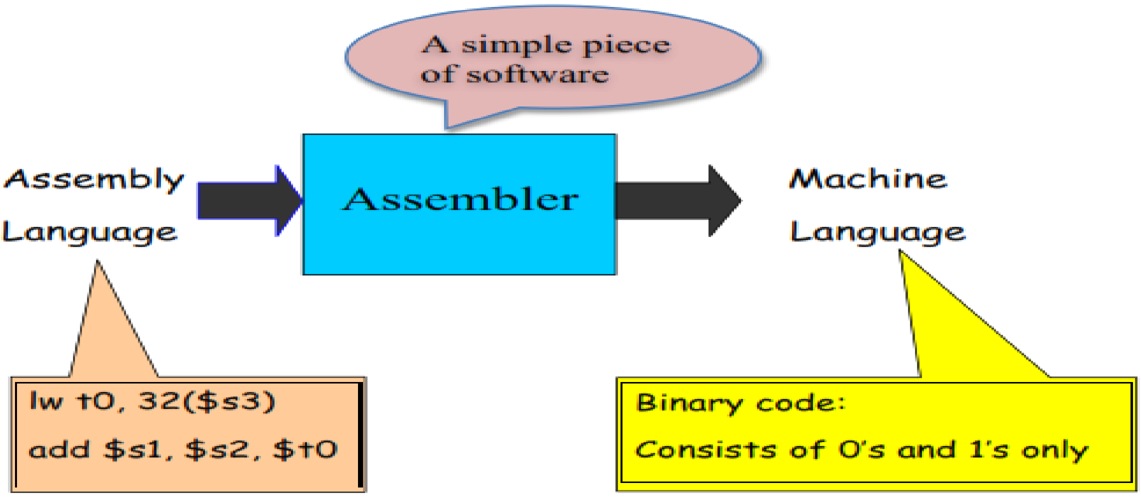
Registers Used in This Chapter



$8	$t0
$9	$t1
$10	$t2
$11	$t3
$12	$t4
$13	$t5
$14	$t6
$15	$t7
$16	$s0
$17	$s1
$18	$s2
$19	$s3
$20	$s4
$21	$s5
$22	$s6
$23	$s7
$24	$t8
$25	$t9

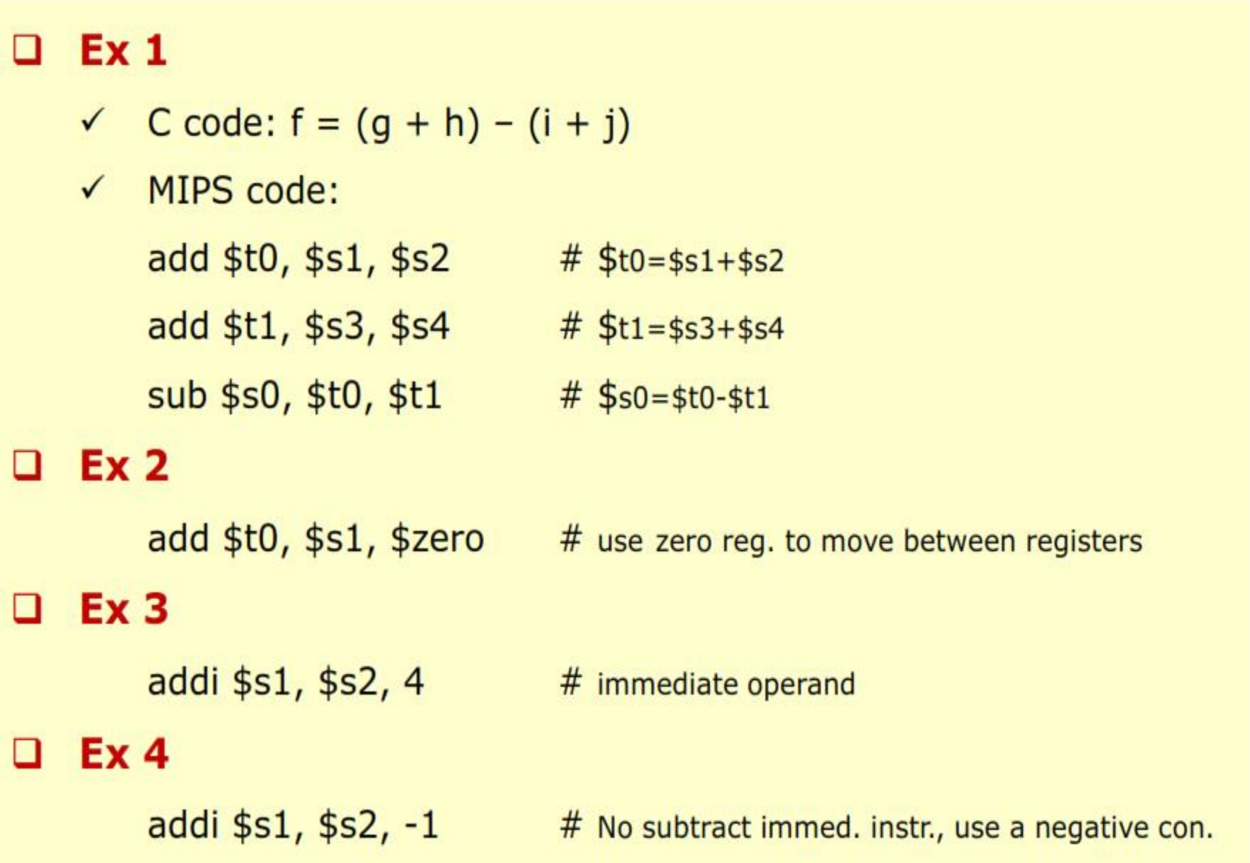
Assembly language programs

What is an Assembler?



```
lw t0, 32($s3)
add $s1, $s2, $t0
```

Register Operands



- □ **Ex 1**
  - ✓ C code: f = (g + h) – (i + j)
  - ✓ MIPS code:

```
add $t0, $s1, $s2       # $t0=$s1+$s2
add $t1, $s3, $s4       # $t1=$s3+$s4
sub $s0, $t0, $t1       # $s0=$t0-$t1
```

- □ **Ex 2**

```
add $t0, $s1, $zero     # use zero reg. to move between registers
```

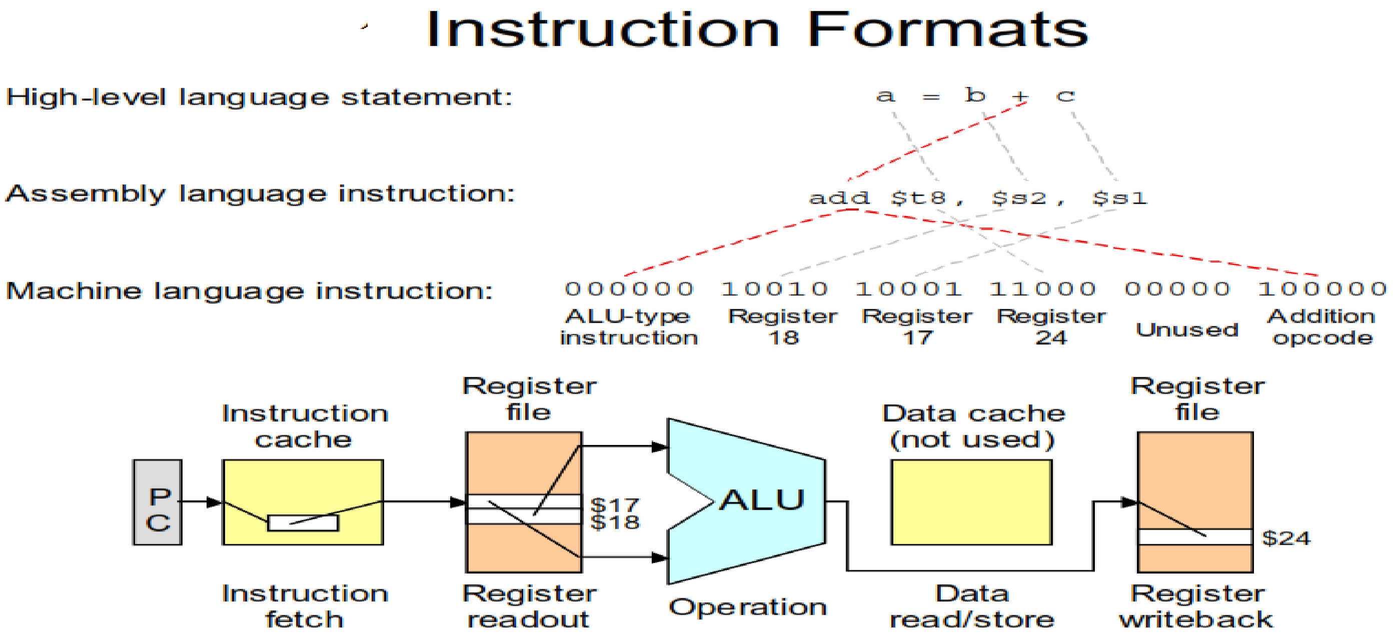
- □ **Ex 3**

```
addi $s1, $s2, 4        # immediate operand
```

- □ **Ex 4**

```
addi $s1, $s2, -1       # No subtract immed. instr., use a negative con.
```

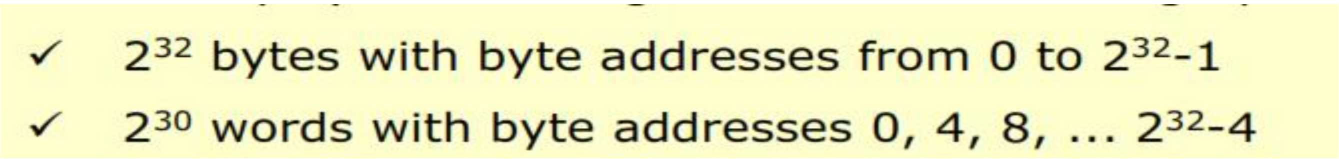
Note: Note that "sub" has a similar format.



Instruction Formats

Memory Operands

- **Main memory used for composite data**
  - Arrays, structures, dynamic data
- **To apply arithmetic operations**
  - Load values from memory into registers
  - Store result from register to memory
  - Load word has destination first, store word has destination last
- **Memory is byte addressed**
  - Each address identifies an 8-bit byte. (memory accessed by a load/store is a byte). i.e. loading words but addressing bytes

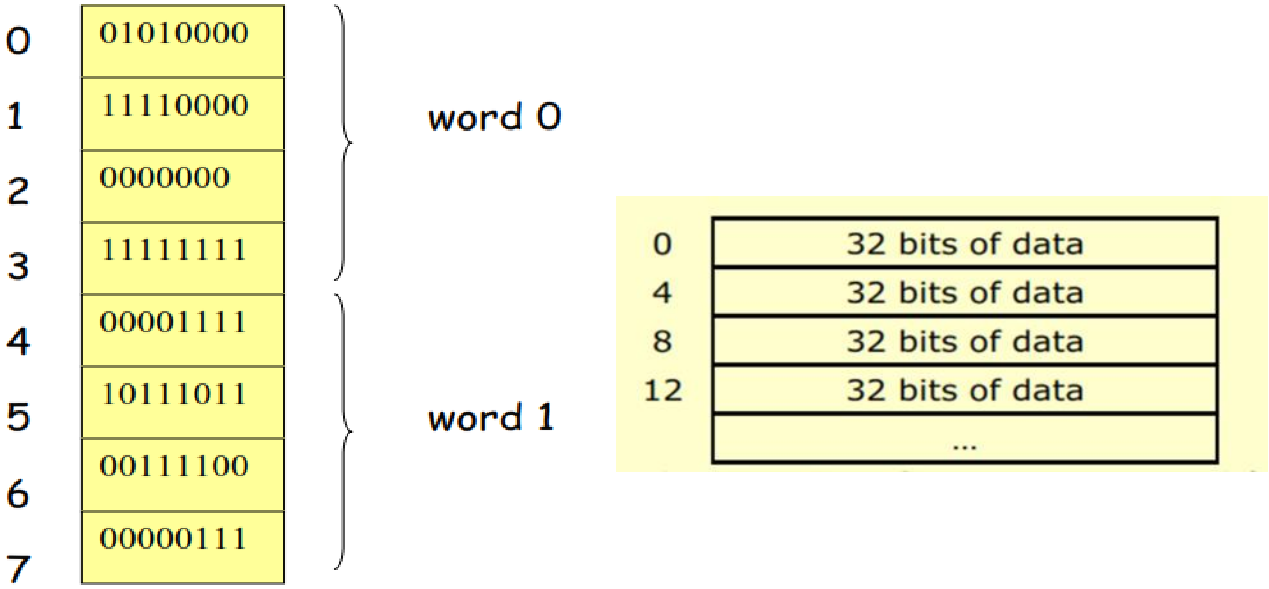


✓ 2^{32} bytes with byte addresses from 0 to 2^{32}-1

✓ 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4

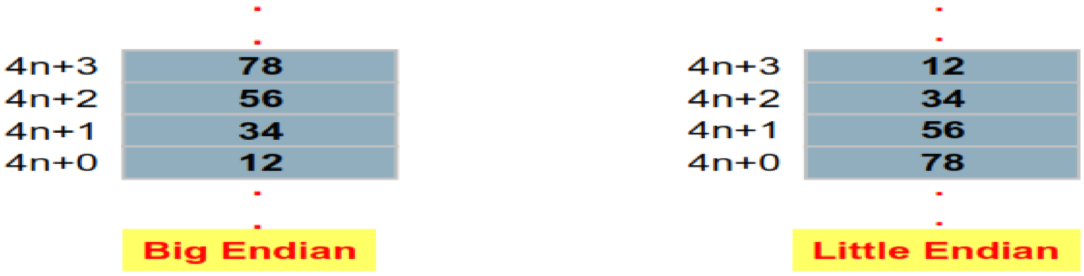
- **Words are aligned in memory**
  - Address must be a multiple of 4

- In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, …, as shown. We say that the word locations have ***aligned addresses*** .



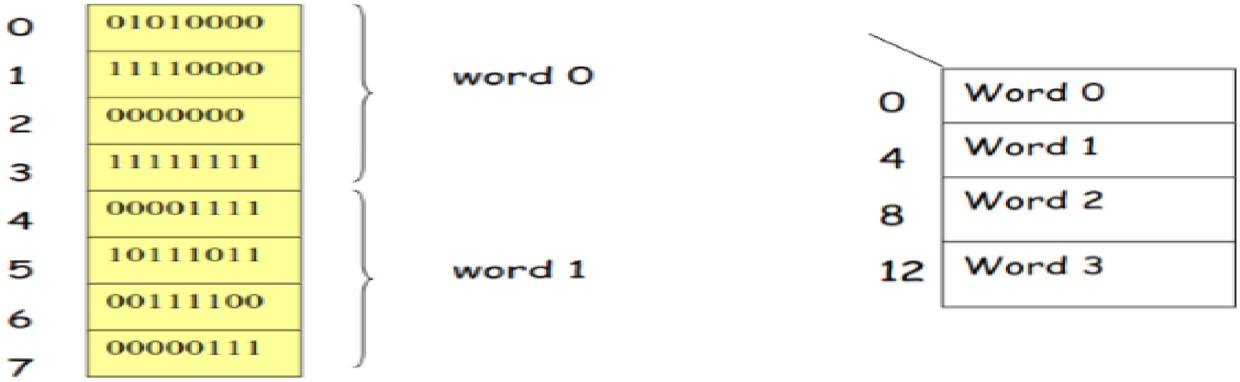
- **Byte Order (Big Endian and Little Endian) an**
  - **Big Endian Byte Order:** The most significant byte (the "**big end**") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory **<u>(MIPS)</u>**
  - **Little Endian Byte Order**: The least significant byte (the "**little end**") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory **(Intel 80x86)**.

- Example
  - In C, `int num = 0x12345678; // a 32-bit word,`
  - how is `num` stored in memory?



- ✓ MIPS is *Big Endian* so bytes in a word is numbered as follows:
  - ✓ byte 0 at the leftmost (most-significant) to byte 3 at the rightmost (least-significant).

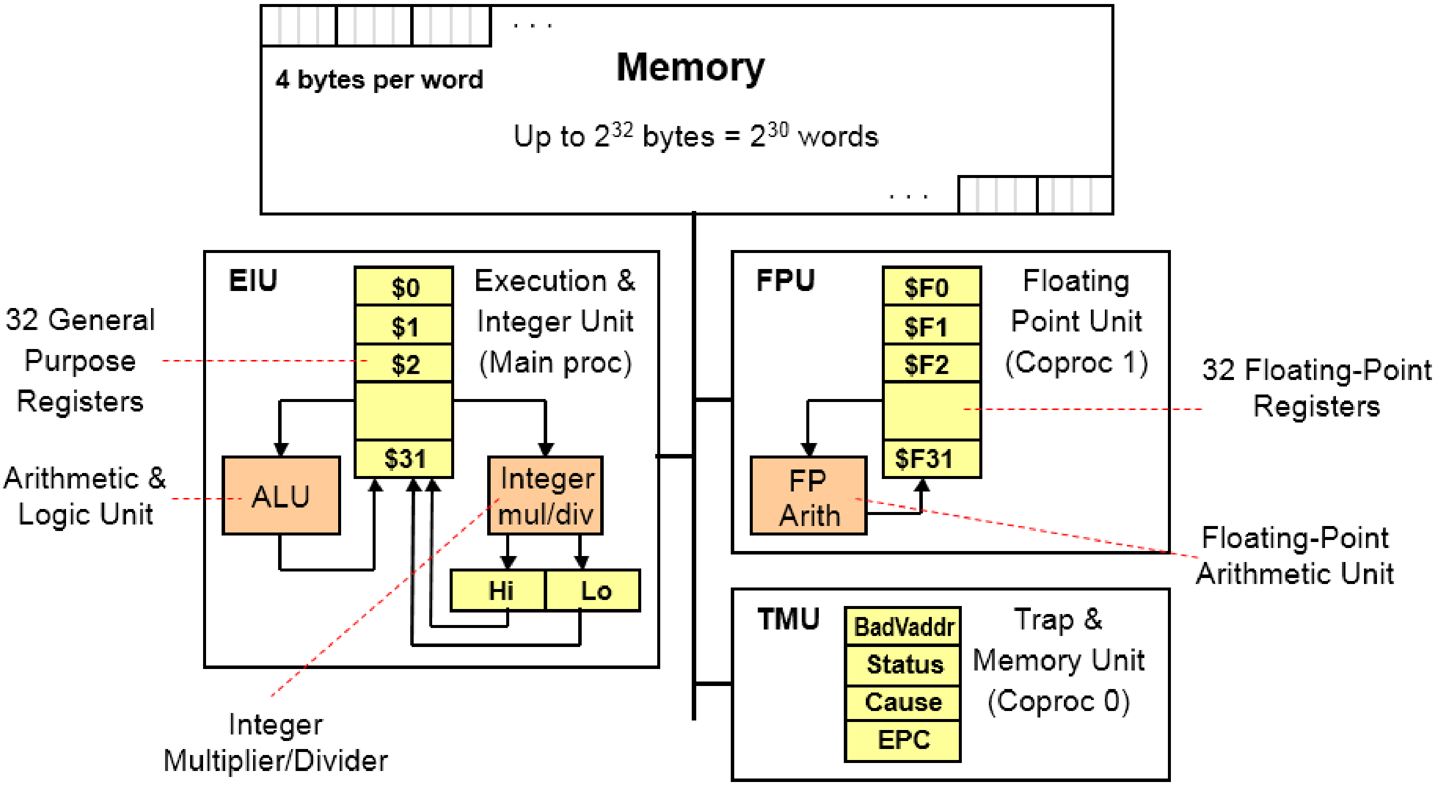
Word 0	Byte 0	Byte 1	Byte 2	Byte 3
Word 1	Byte 4	Byte 5	Byte 6	Byte 7
...	...	...	...	...



0	01010000
1	11110000
2	0000000
3	11111111
4	00001111
5	10111011
6	00111100
7	00000111

0	Word 0
4	Word 1
8	Word 2
12	Word 3

Logical View of the MIPS Processor

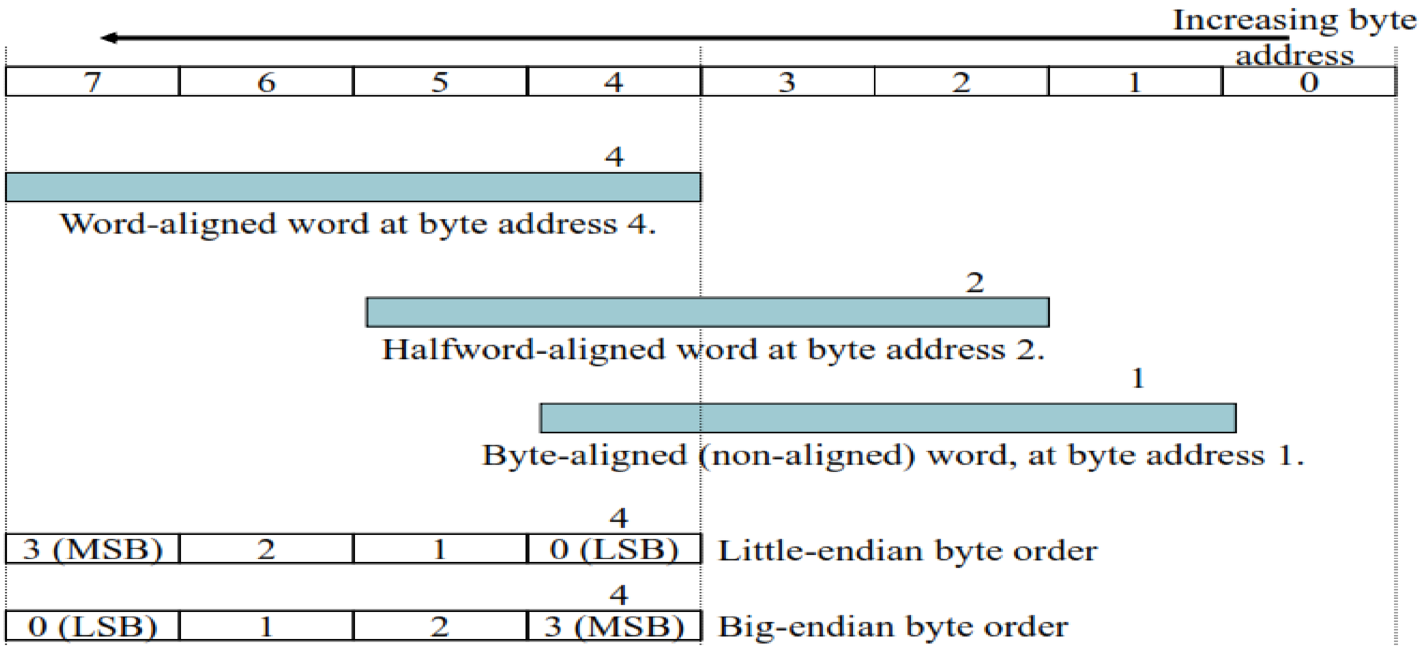

. . .
Memory
4 bytes per word
Up to 2^{32} bytes = 2^{30} words
. . .
EIU
$0
$1
$2
$31
Execution & Integer Unit (Main proc)
32 General Purpose Registers
ALU
Arithmetic & Logic Unit
Integer mul/div
Hi
Lo
Integer Multiplier/Divider
FPU
$F0
$F1
$F2
$F31
Floating Point Unit (Coproc 1)
32 Floating-Point Registers
FP Arith
Floating-Point Arithmetic Unit
TMU
BadVaddr
Status
Cause
EPC
Trap & Memory Unit (Coproc 0)

Memory Alignment

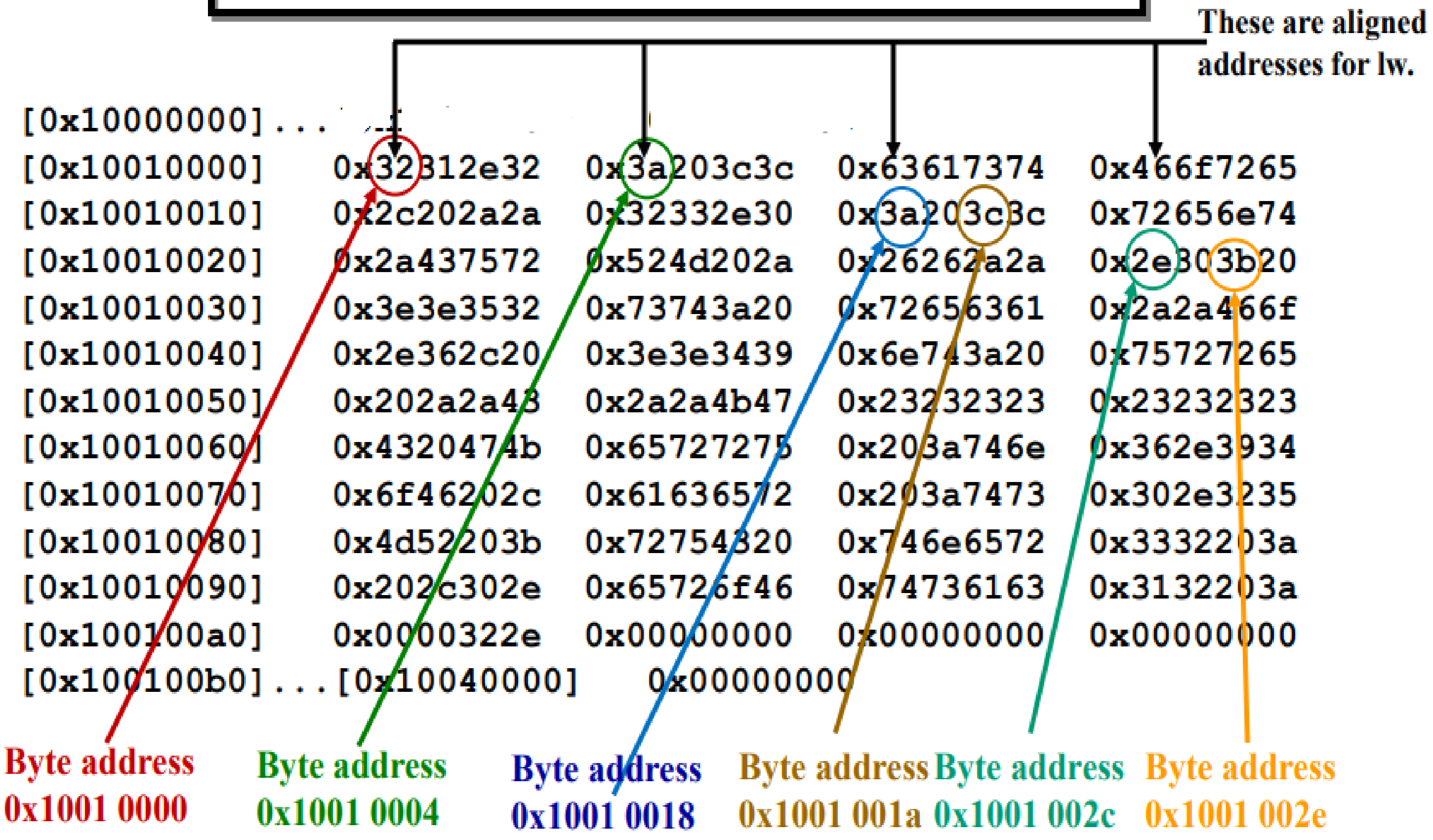
- The memory is typically aligned on a word or double-word boundary.
- An access to object of size S bytes at byte address A is called aligned if $A \bmod S = 0$.
- Access to an unaligned operand may require more memory accesses !!

Memory Alignment on different architectures

Memory address	Alignment (8 bit)	Alignment (16 bit)	Alignment (32 bit)	Alignment (64 bit)
0x0000_0000	Aligned	Aligned	Aligned	Aligned
0x0000_0001	Aligned	Non Aligned	Non Aligned	Non Aligned
0x0000_0002	Aligned	Aligned	Non Aligned	Non Aligned
0x0000_0003	Aligned	Non Aligned	Non Aligned	Non Aligned
0x0000_0004	Aligned	Aligned	Aligned	Non Aligned
0x0000_0005	Aligned	Non Aligned	Non Aligned	Non Aligned
0x0000_0006	Aligned	Aligned	Non Aligned	Non Aligned
0x0000_0007	Aligned	Non Aligned	Non Aligned	Non Aligned
0x0000_0008	Aligned	Aligned	Aligned	Aligned


Increasing byte address
7
6
5
4
3
2
1
0
4
Word-aligned word at byte address 4.
2
Halfword-aligned word at byte address 2.
1
Byte-aligned (non-aligned) word, at byte address 1.
4
3 (MSB)
2
1
0 (LSB)
Little-endian byte order
4
0 (LSB)
1
2
3 (MSB)
Big-endian byte order

SPIM Data Window Readout



These are aligned addresses for lw.

```
[0x10000000]...
[0x10010000]    0x32312e32   0x3a203c3c   0x63617374   0x466f7265
[0x10010010]    0x2c202a2a   0x32332e30   0x3a203c3c   0x72656e74
[0x10010020]    0x2a437572   0x524d202a   0x26262a2a   0x2e303b20
[0x10010030]    0x3e3e3532   0x73743a20   0x72656361   0x2a2a466f
[0x10010040]    0x2e362c20   0x3e3e3439   0x6e743a20   0x75727265
[0x10010050]    0x202a2a43   0x2a2a4b47   0x23232323   0x23232323
[0x10010060]    0x4320474b   0x65727275   0x203a746e   0x362e3934
[0x10010070]    0x6f46202c   0x61636572   0x203a7473   0x302e3235
[0x10010080]    0x4d52203b   0x72754320   0x746e6572   0x3332203a
[0x10010090]    0x202c302e   0x65726f46   0x74736163   0x3132203a
[0x100100a0]    0x0000322e   0x00000000   0x00000000   0x00000000
[0x100100b0]...[0x10040000]     0x00000000
```

Byte address 0x1001 0000

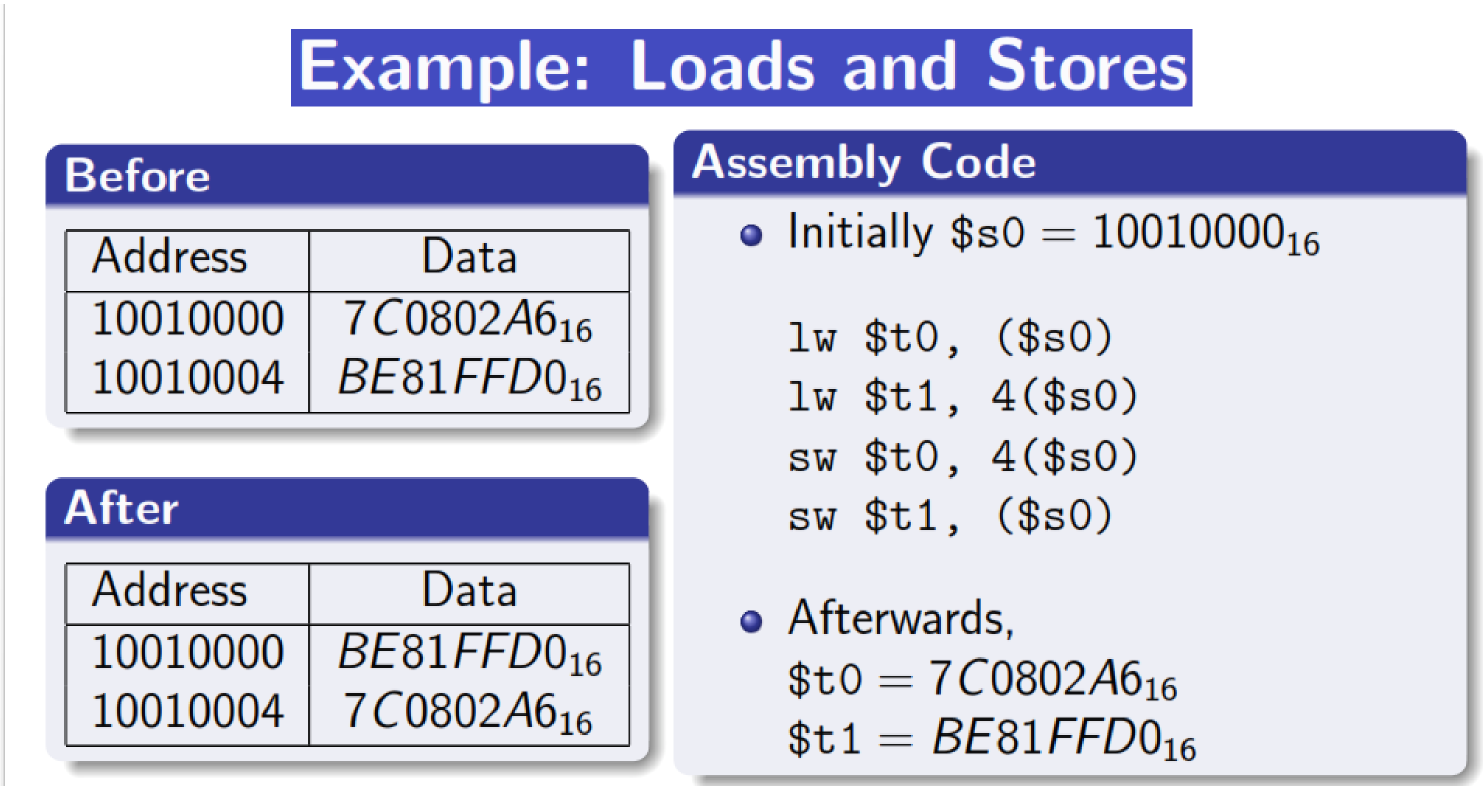
Byte address 0x1001 0004

Byte address 0x1001 0018

Byte address 0x1001 001a

Byte address 0x1001 002c

Byte address 0x1001 002e



Example: Loads and Stores

Before

Address	Data
10010000	$7C0802A6_{16}$
10010004	$BE81FFD0_{16}$

After

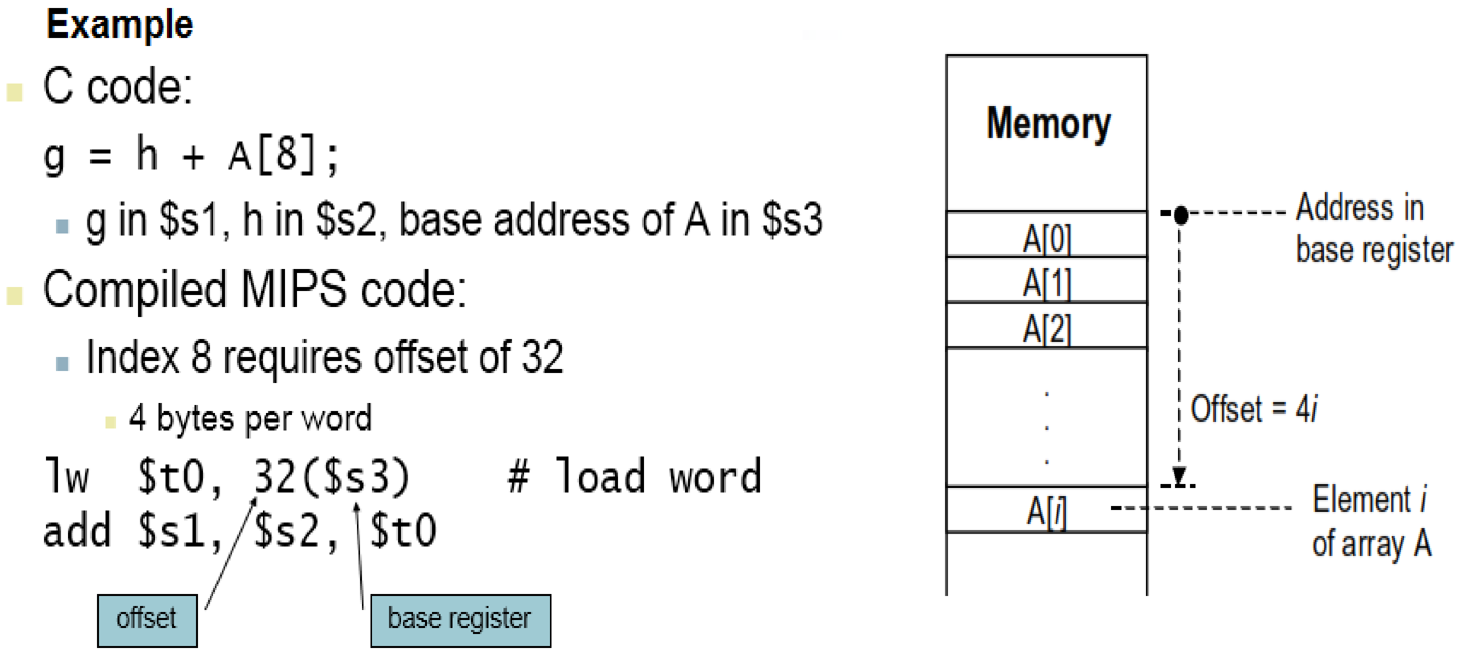
Address	Data
10010000	$BE81FFD0_{16}$
10010004	$7C0802A6_{16}$

Assembly Code

- Initially `$s0` $= 10010000_{16}$

```
lw $t0, ($s0)
lw $t1, 4($s0)
sw $t0, 4($s0)
sw $t1, ($s0)
```

- Afterwards,
  `$t0` $= 7C0802A6_{16}$
  `$t1` $= BE81FFD0_{16}$



Example

- C code:

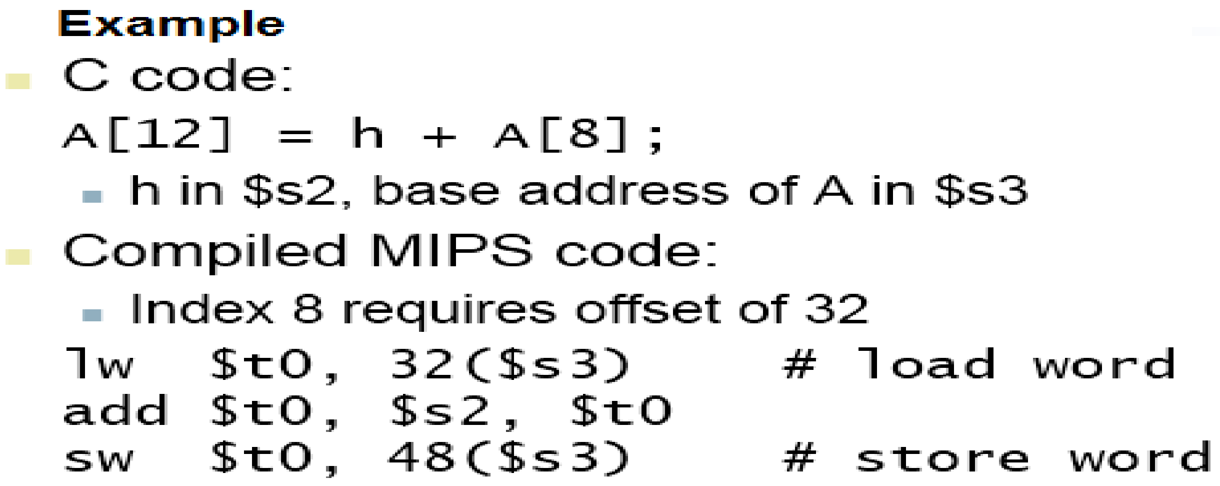
  `g = h + A[8];`

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register



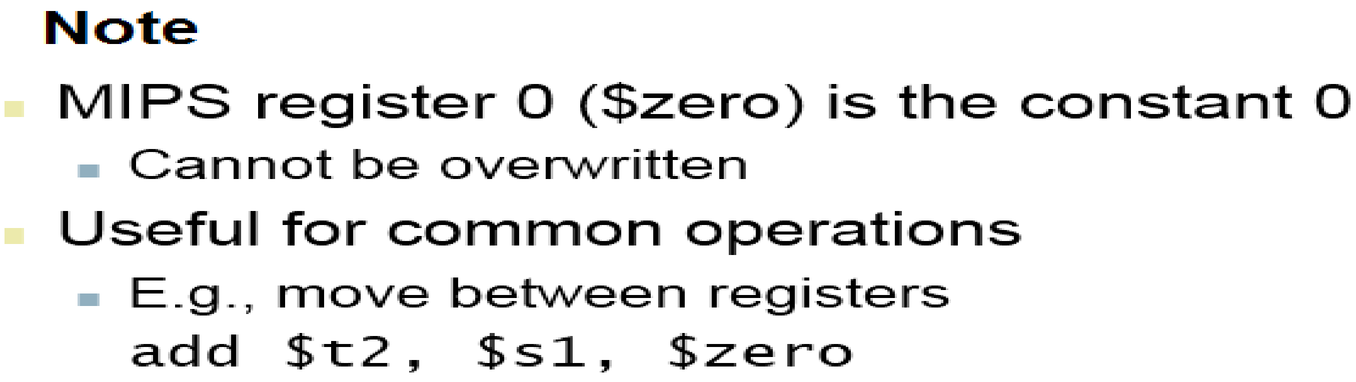
Example

- C code:

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

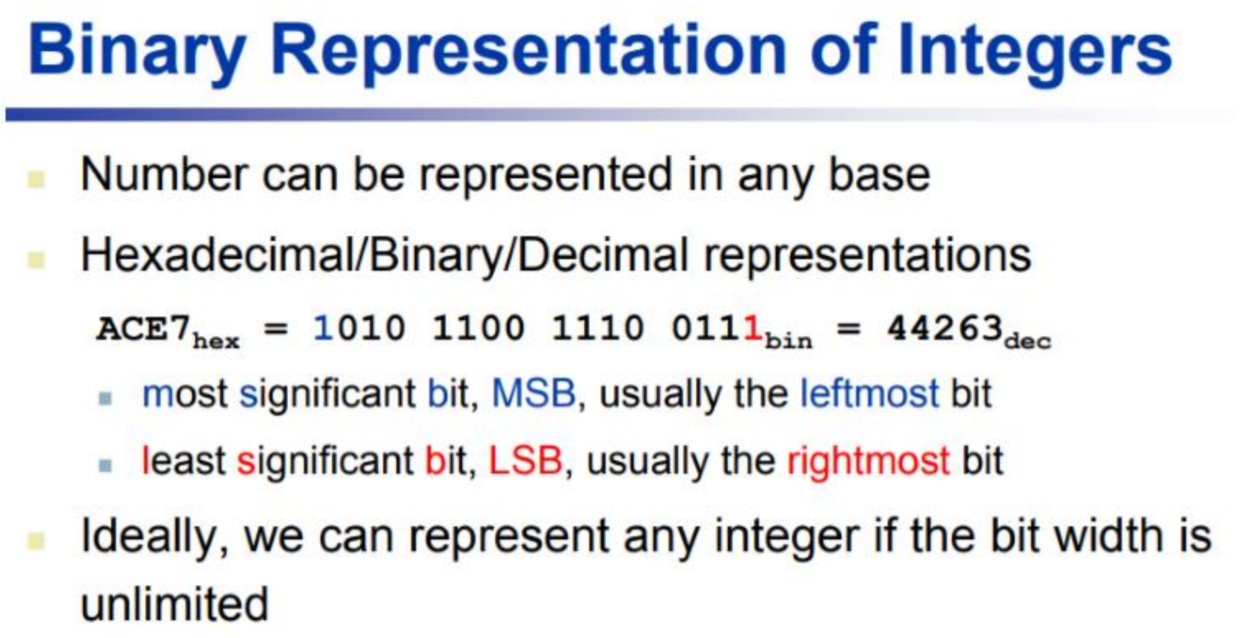
```
lw   $t0, 32($s3)     # load word
add  $t0, $s2, $t0
sw   $t0, 48($s3)     # store word
```



Note

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

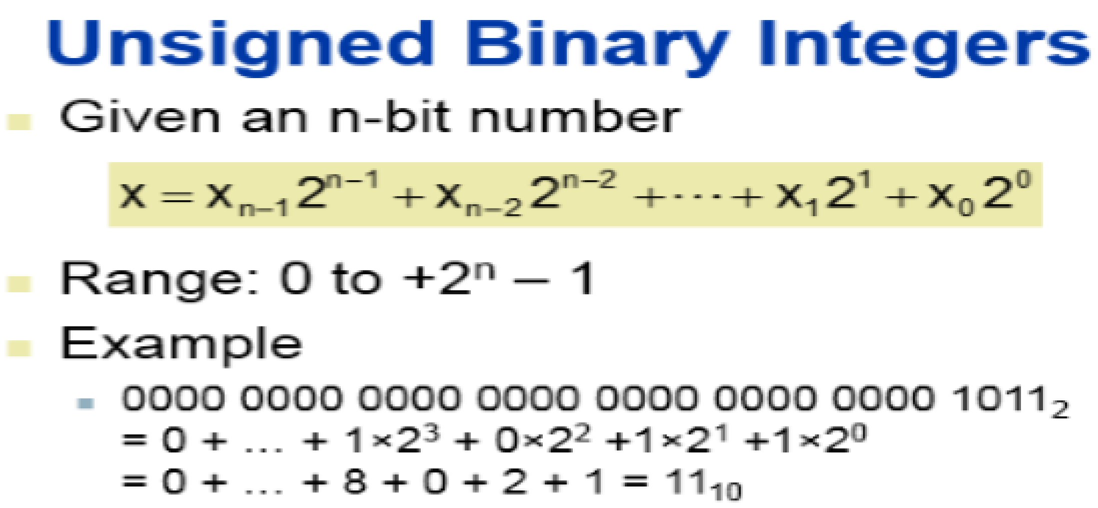


Binary Representation of Integers

- Number can be represented in any base
- Hexadecimal/Binary/Decimal representations

  $\mathtt{ACE7}_{hex} = \mathtt{1010\ 1100\ 1110\ 0111}_{bin} = \mathtt{44263}_{dec}$

  - most significant bit, MSB, usually the leftmost bit
  - least significant bit, LSB, usually the rightmost bit
- Ideally, we can represent any integer if the bit width is unlimited

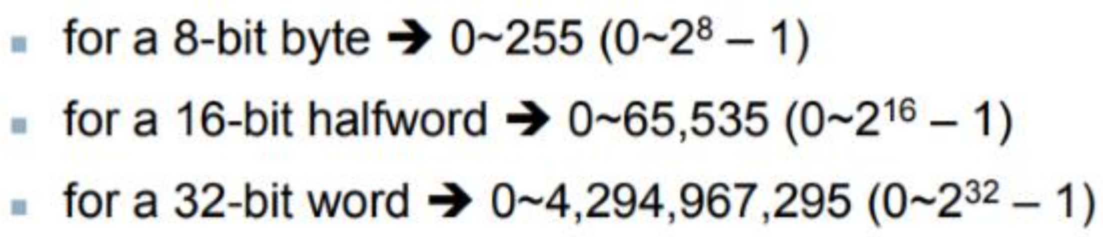


Unsigned Binary Integers

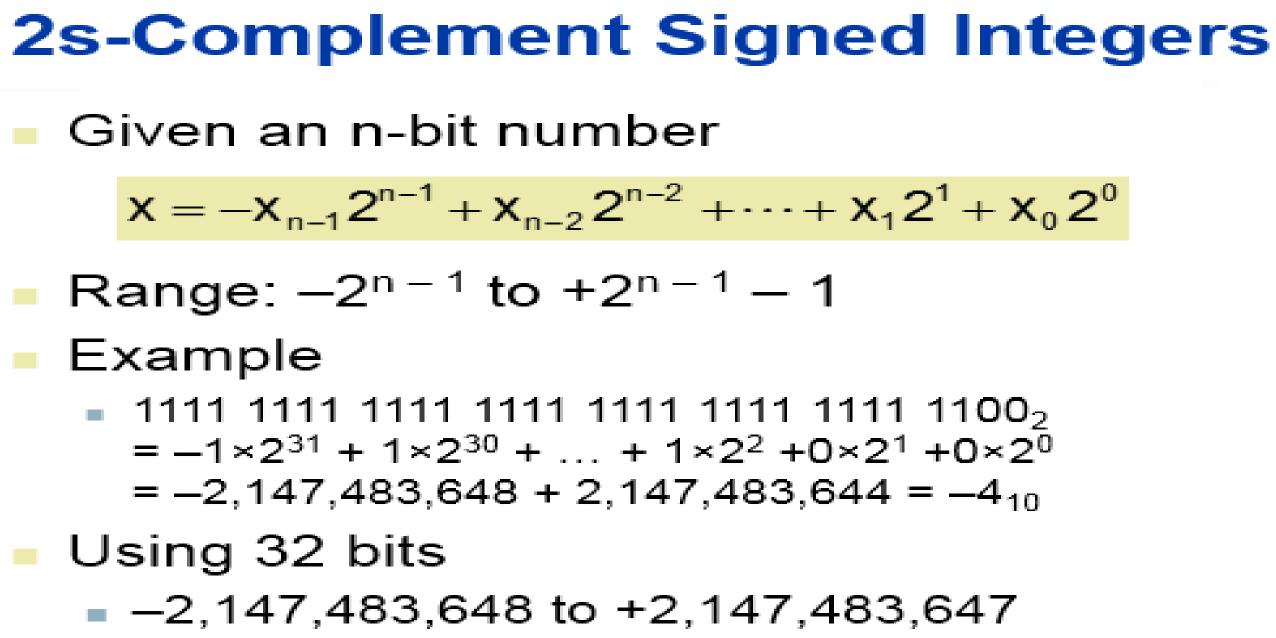
- Given an n-bit number

  $$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$



- for a 8-bit byte ➔ 0~255 ($0\sim2^8 - 1$)
- for a 16-bit halfword ➔ 0~65,535 ($0\sim2^{16} - 1$)
- for a 32-bit word ➔ 0~4,294,967,295 ($0\sim2^{32} - 1$)

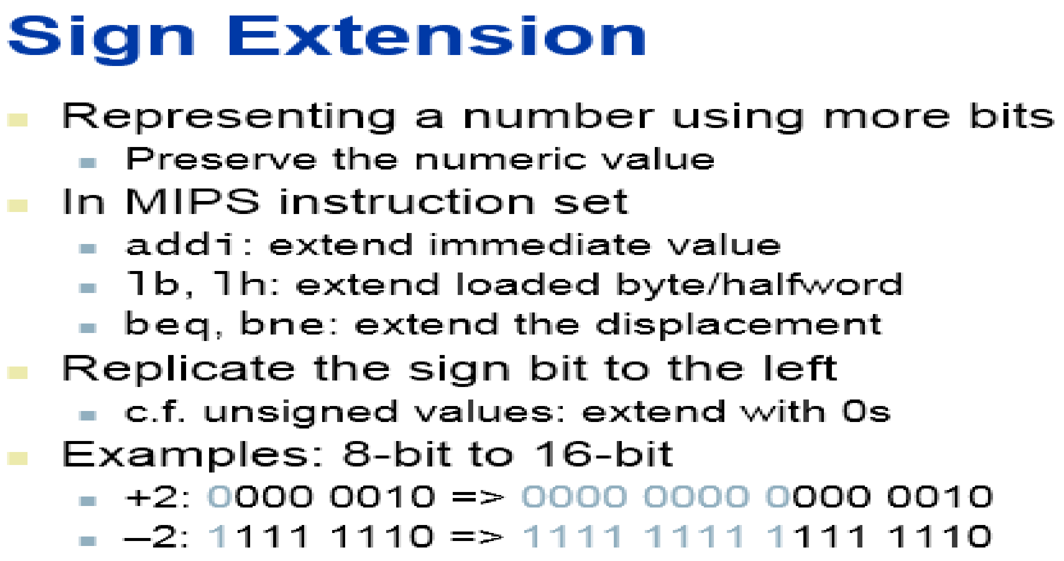


2s-Complement Signed Integers

- Given an n-bit number

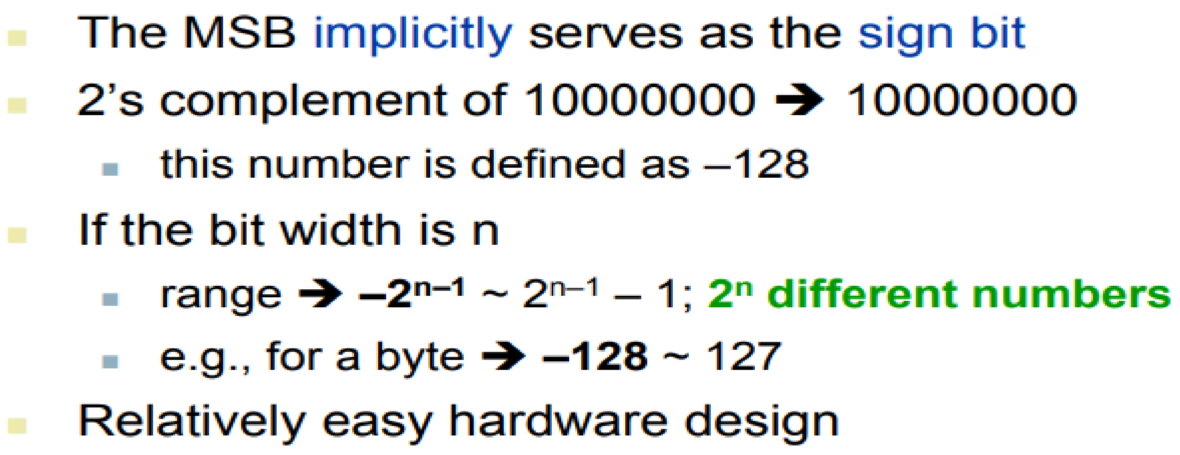
$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647



Sign Extension

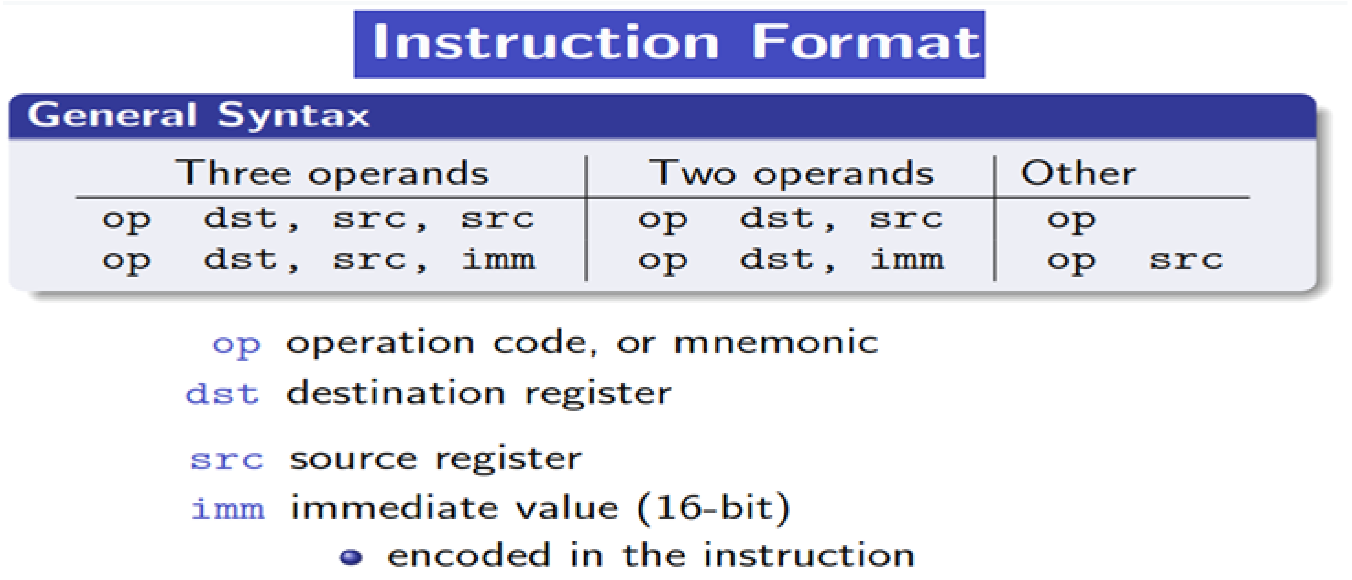
- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - addi: extend immediate value
  - lb, lh: extend loaded byte/halfword
  - beq, bne: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110



- The MSB implicitly serves as the sign bit
- 2's complement of 10000000 → 10000000
  - this number is defined as –128
- If the bit width is n
  - range → $-2^{n-1} \sim 2^{n-1} - 1$; 2^n different numbers
  - e.g., for a byte → **–128** ~ 127
- Relatively easy hardware design

MIPS Instruction Formats

- All MIPS instructions are encoded in binary.
- All MIPS instructions are 32 bits long.
- All instructions have: op (or opcode): operation code (specifies the operation) (first 6 bits)



Instruction Format

General Syntax

Three operands	Two operands	Other
op dst, src, src	op dst, src	op
op dst, src, imm	op dst, imm	op src

op operation code, or mnemonic

dst destination register

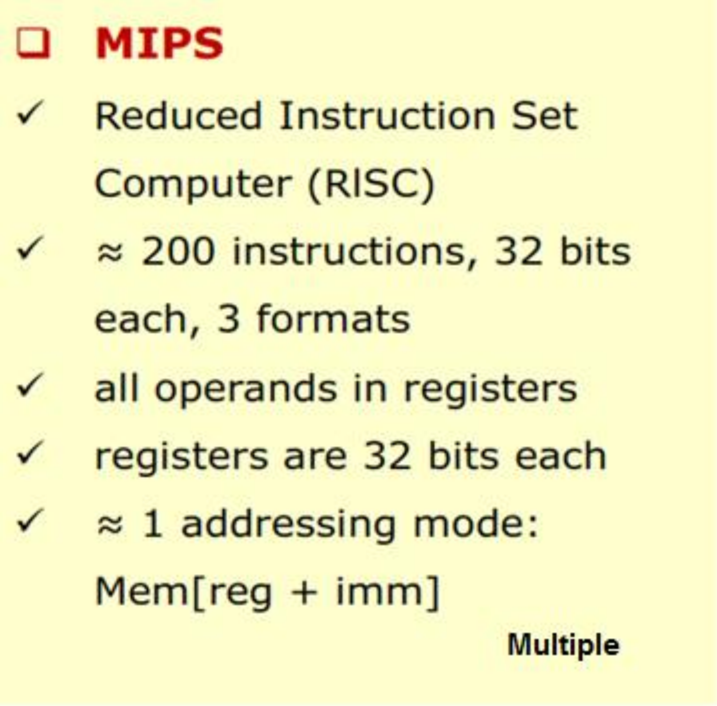
src source register

imm immediate value (16-bit)

- encoded in the instruction

- There are ***three instruction categories***:

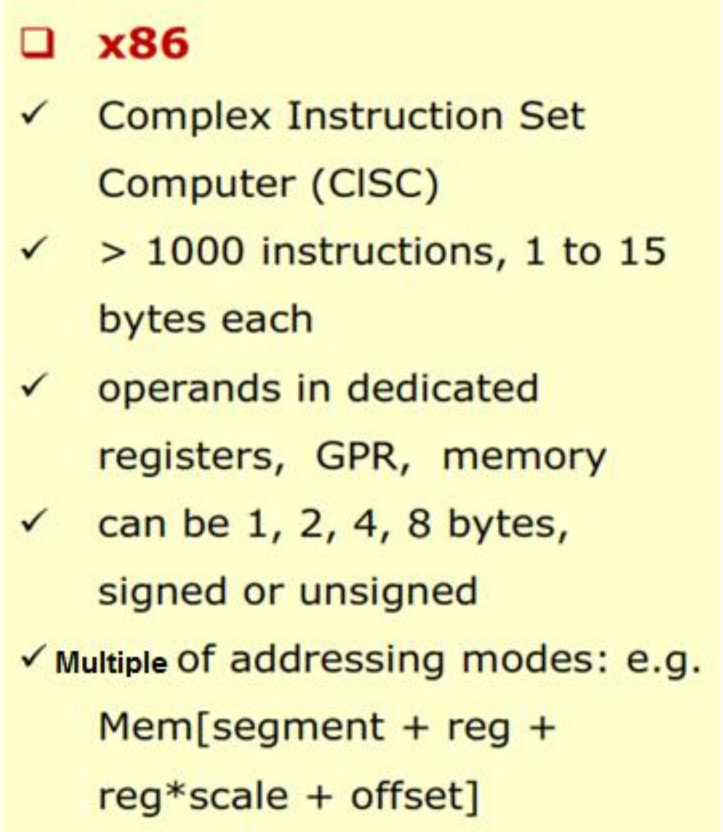
```
                  6         5         5         5        5        6
               +---------+---------+--------+--------+-------+---------+
R-type format| Op-code|      Rs   |   Rt   |  Rd     |  SA   |Funct-code|
               +---------+---------+--------+--------+-------+---------+

                  6         5         5                  16
               +---------+---------+--------+---------------------------+
I-type format|Op-code |      Rs |    Rt   | 2's complement constant|
               +---------+---------+--------+---------------------------+

                 6                           26
               +---------+--------------------------------------------+
J-type format| Op-code|                 jump_target                 |
               +---------+--------------------------------------------+
                 ^                                                    ^
                 |                                                    |
            bit 31                                              bit 0
```



❑ **MIPS**

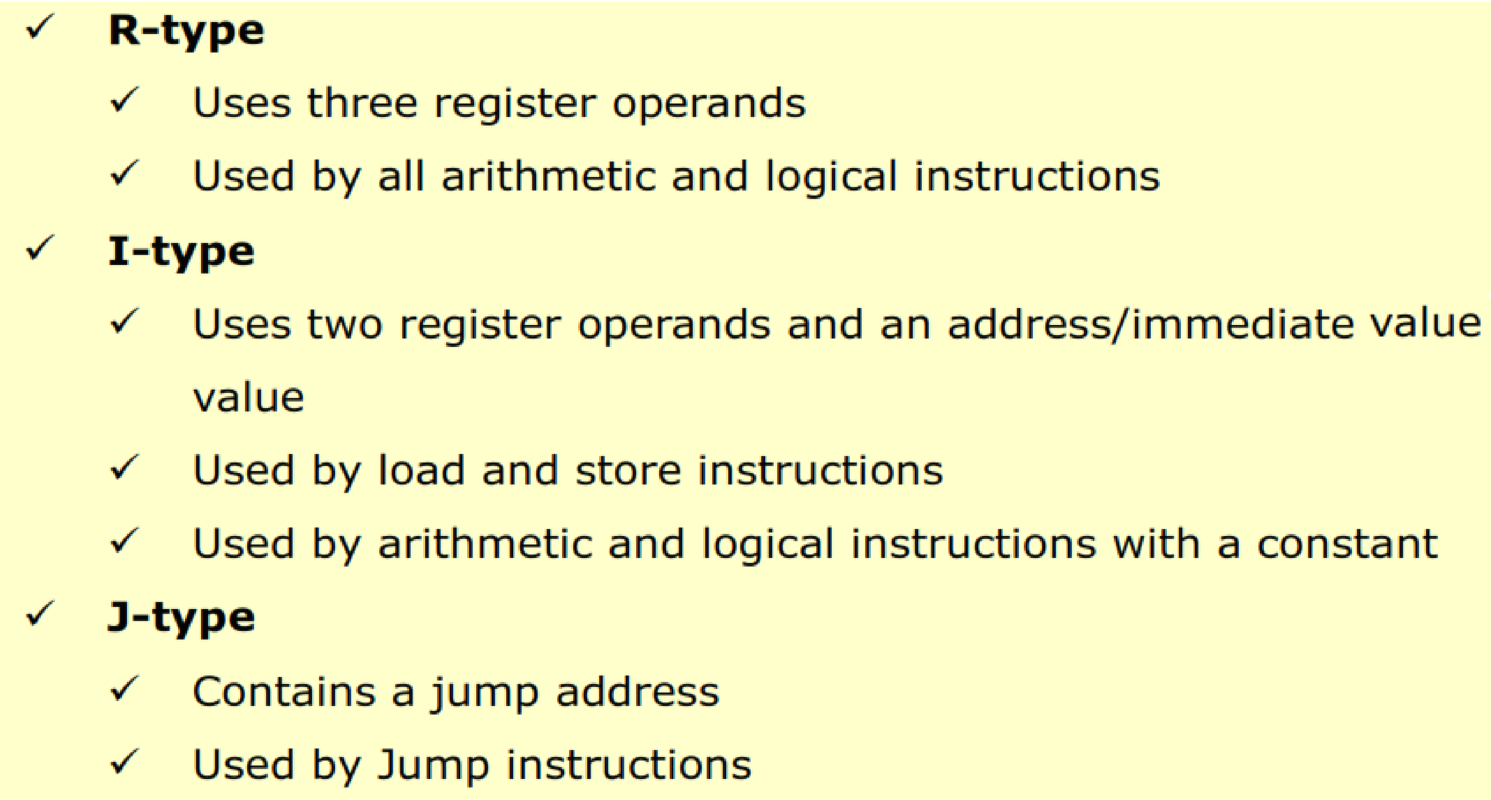
- ✓ Reduced Instruction Set Computer (RISC)
- ✓ ≈ 200 instructions, 32 bits each, 3 formats
- ✓ all operands in registers
- ✓ registers are 32 bits each
- ✓ ≈ 1 addressing mode: Mem[reg + imm]

Multiple

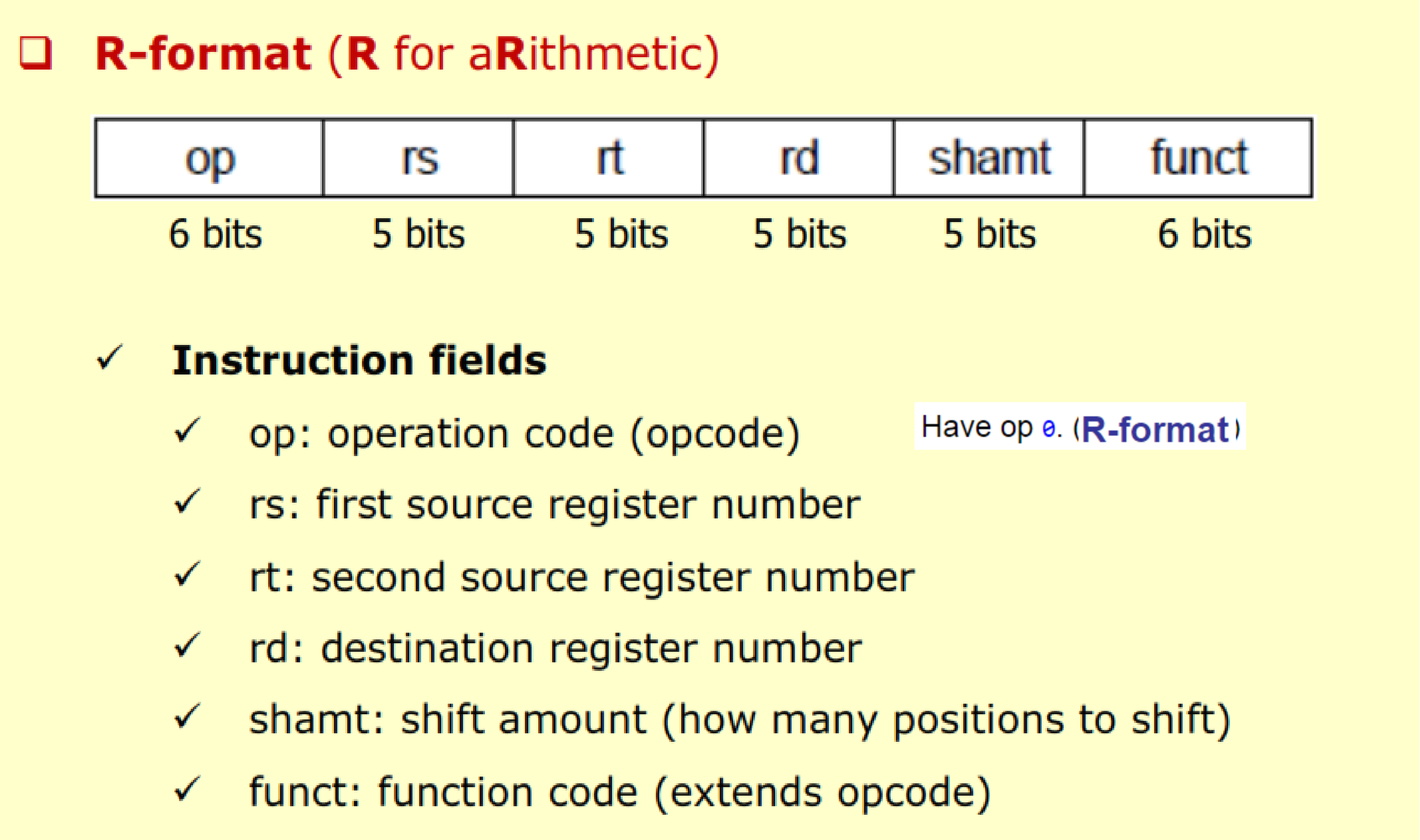


❑ **x86**

- ✓ Complex Instruction Set Computer (CISC)
- ✓ > 1000 instructions, 1 to 15 bytes each
- ✓ operands in dedicated registers, GPR, memory
- ✓ can be 1, 2, 4, 8 bytes, signed or unsigned
- ✓ **Multiple** of addressing modes: e.g. Mem[segment + reg + reg*scale + offset]



- ✓ **R-type**
  - ✓ Uses three register operands
  - ✓ Used by all arithmetic and logical instructions
- ✓ **I-type**
  - ✓ Uses two register operands and an address/immediate value value
  - ✓ Used by load and store instructions
  - ✓ Used by arithmetic and logical instructions with a constant
- ✓ **J-type**
  - ✓ Contains a jump address
  - ✓ Used by Jump instructions

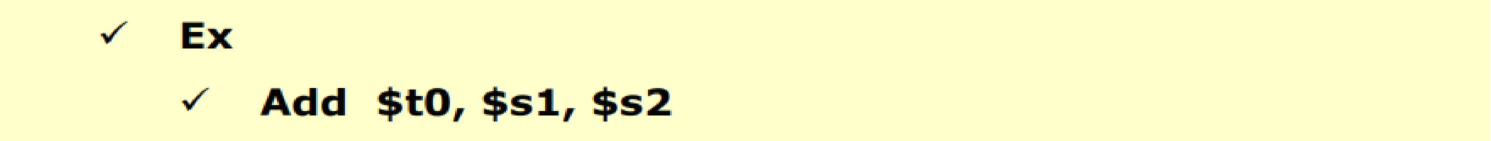


❑ **R-format** (**R** for a**R**ithmetic)

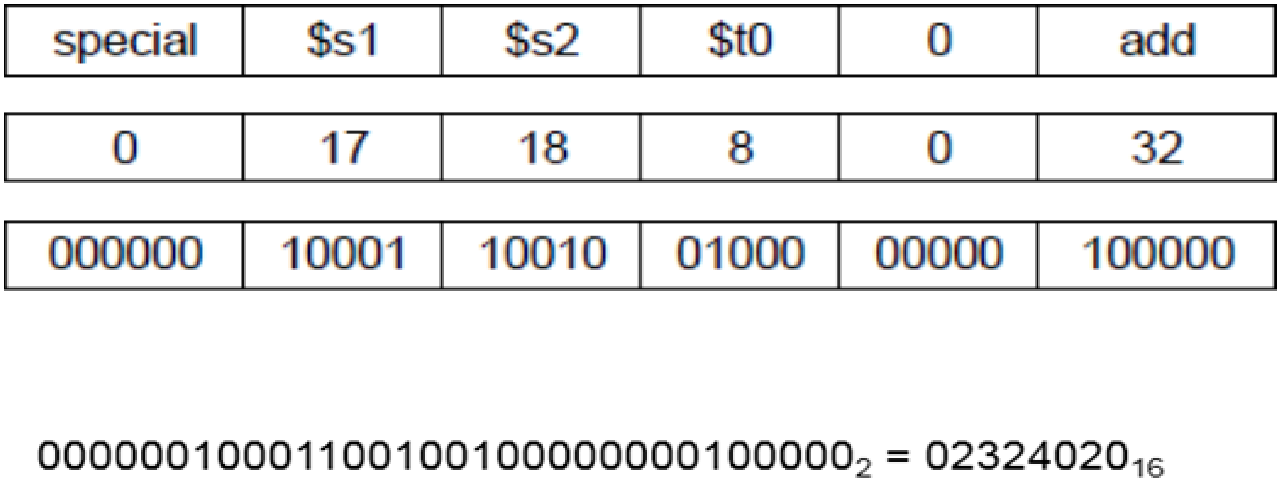
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- ✓ **Instruction fields**
  - ✓ op: operation code (opcode)
  - ✓ rs: first source register number
  - ✓ rt: second source register number
  - ✓ rd: destination register number
  - ✓ shamt: shift amount (how many positions to shift)
  - ✓ funct: function code (extends opcode)

Have op 0. (**R-format**)

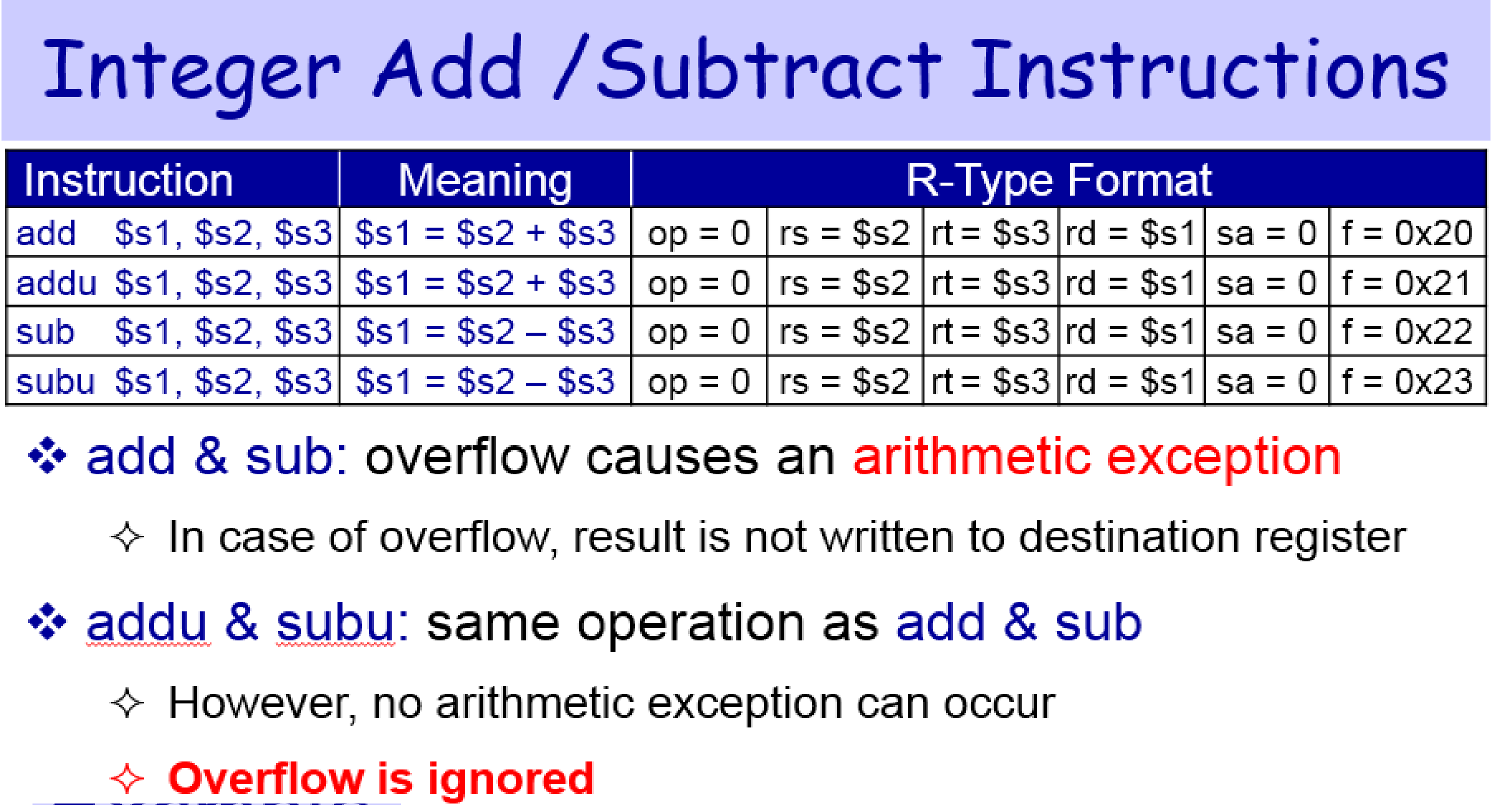


- ✓ **Ex**
  - ✓ **Add $t0, $s1, $s2**



special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

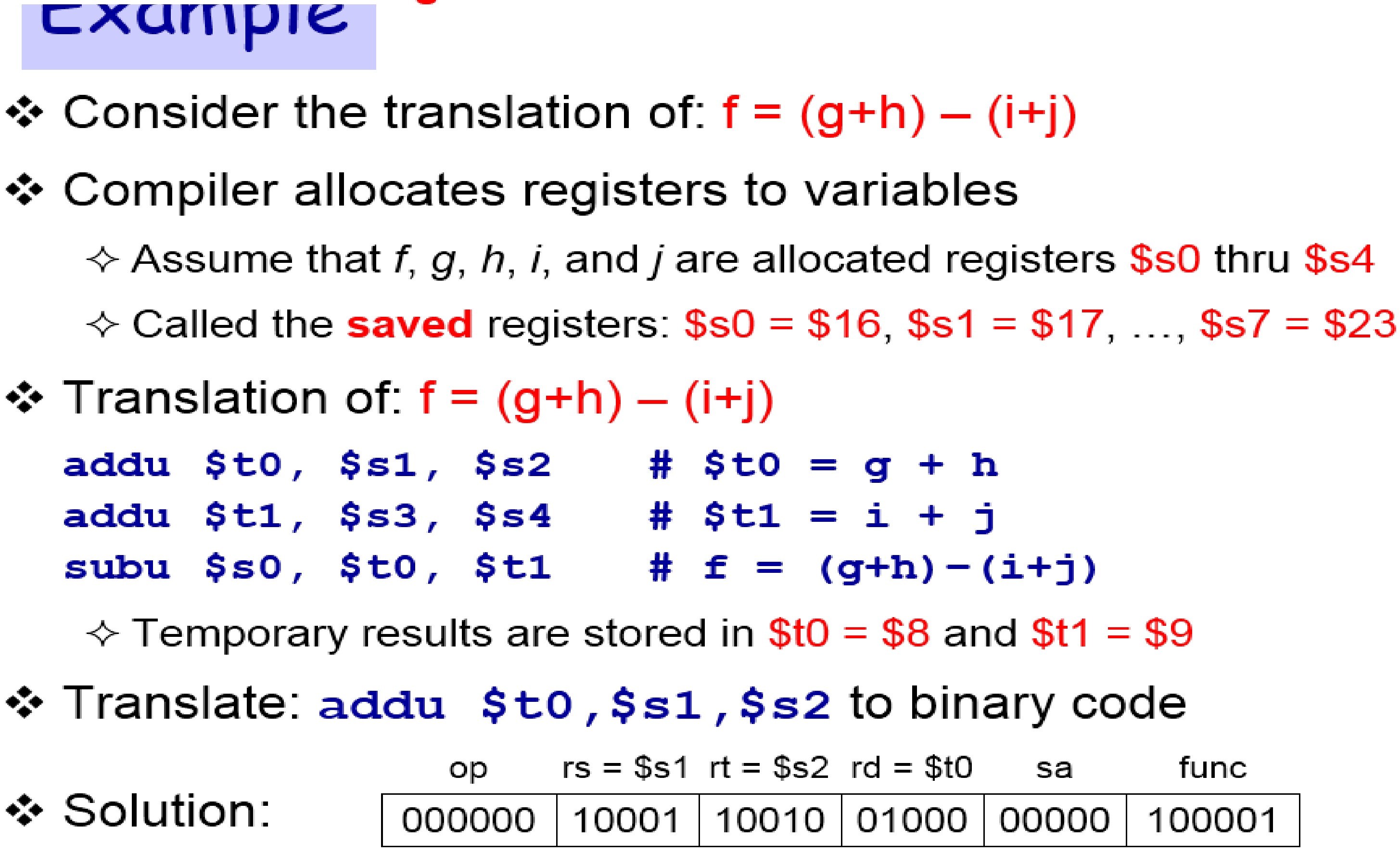
$00000010001100100100000000100000_2 = 02324020_{16}$



Integer Add /Subtract Instructions

Instruction	Meaning	R-Type Format					
add $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x20
addu $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x21
sub $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x22
subu $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x23

- add & sub: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register
- addu & subu: same operation as add & sub
  - However, no arithmetic exception can occur
  - **Overflow is ignored**



Example

- Consider the translation of: f = (g+h) – (i+j)
- Compiler allocates registers to variables
  - Assume that *f*, *g*, *h*, *i*, and *j* are allocated registers $s0 thru $s4
  - Called the **saved** registers: $s0 = $16, $s1 = $17, …, $s7 = $23
- Translation of: f = (g+h) – (i+j)

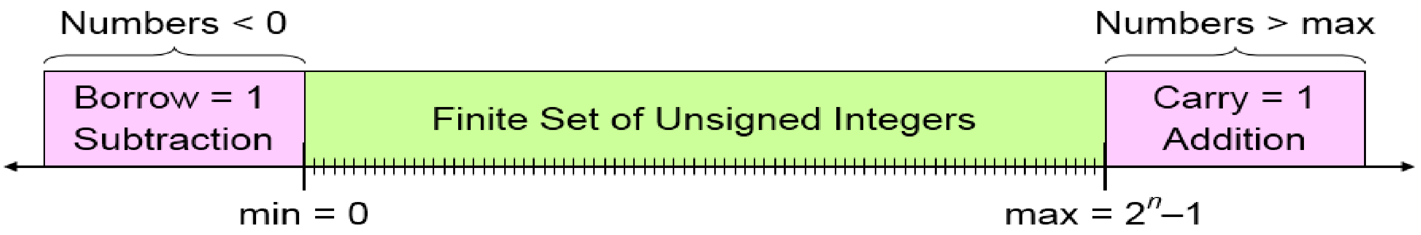
```
addu  $t0, $s1, $s2     # $t0 = g + h
addu  $t1, $s3, $s4     # $t1 = i + j
subu  $s0, $t0, $t1     # f = (g+h)–(i+j)
```

  - Temporary results are stored in $t0 = $8 and $t1 = $9
- Translate: `addu $t0,$s1,$s2` to binary code
- Solution:

op	rs = $s1	rt = $s2	rd = $t0	sa	func
000000	10001	10010	01000	00000	100001

Range, Carry, Borrow, and Overflow

- Bits have NO meaning. The same *n* bits stored in a register can represent an unsigned or a signed integer.
- Unsigned Integers: *n*-bit representation



Numbers < 0

Numbers > max

Borrow = 1 Subtraction

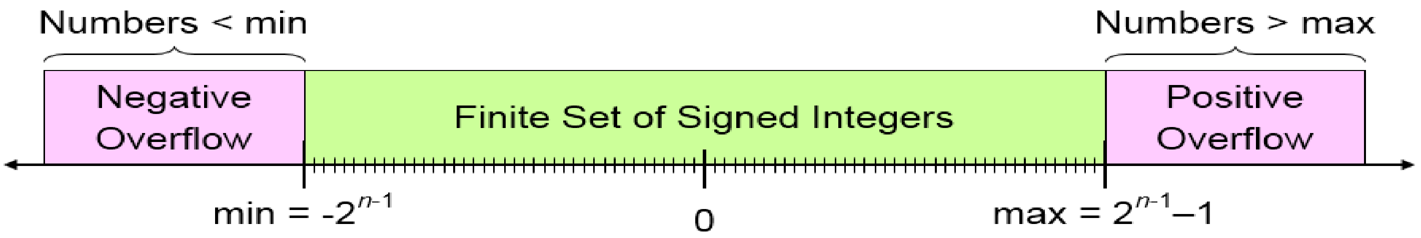
Finite Set of Unsigned Integers

Carry = 1 Addition

min = 0

max = $2^n – 1$

- Signed Integers: *n*-bit 2's complement representation



Numbers < min

Numbers > max

Negative Overflow

Finite Set of Signed Integers

Positive Overflow

min = -2^{n-1}

0

max = $2^{n-1} – 1$

Carry and Overflow

- Carry is useful when adding (subtracting) unsigned integers
  - Carry indicates that the **unsigned sum is out of range**
- Overflow is useful when adding (subtracting) signed integers
  - Overflow indicates that the **signed sum is out of range**
- Range for 32-bit unsigned integers = 0 to (2^{32} – 1)
- Range for 32-bit signed integers = -2^{31} to (2^{31} – 1)
- Example 1: Carry = 1, Overflow = 0 (NO overflow)

```
 11111 1            1 11     1
  1000 0100 0000 0000 1110 0001 0100 0001
+ 1111 1111 0000 0000 1111 0101 0010 0000
-----------------------------------------
  1000 0011 0000 0001 1101 0110 0110 0001
```

Unsigned sum is out-of-range, but the Signed sum is correct

- Example 2: Carry = 0, Overflow = 1

```
01111 1               11    1
 0010 0100 0000 0100 1011 0001 0100 0100
+0111 1111 0111 0000 0011 0101 0000 0010
 ---------------------------------------
 1010 0011 0111 0100 1110 0110 0100 0110
```

Unsigned sum is correct, but the Signed sum is out-of-range

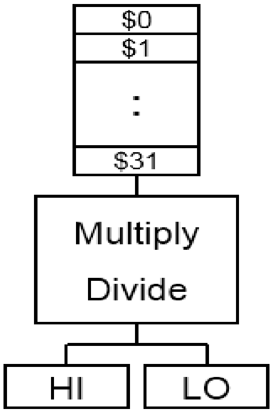
- Example 3: Carry = 1, Overflow = 1

```
1  11 1               11    1
 1000 0100 0000 0100 1011 0001 0100 0100
+1001 1111 0111 0000 0011 0101 0000 0010
 ---------------------------------------
 0010 0011 0111 0100 1110 0110 0100 0110
```

Both the Unsigned and Signed sums are out-of-range

Integer Multiplication & Division

- Consider a×b and a/b where a and b are in $s1 and $s2
  - Signed multiplication: `mult $s1,$s2`
  - Unsigned multiplication: `multu $s1,$s2`
  - Signed division: `div $s1,$s2`
  - Unsigned division: `divu $s1,$s2`
- For multiplication, result is 64 bits
  - LO = low-order 32-bit and HI = high-order 32-bit
- For division
  - LO = 32-bit quotient and HI = 32-bit remainder
  - If divisor is 0 then result is unpredictable
- Moving data
  - `mflo rd` (move from LO to rd), `mfhi rd` (move from HI to rd)
  - `mtlo rs` (move to LO from rs), `mthi rs` (move to HI from rs)



Instruction	Meaning	Format					
mult rs, rt	hi, lo = rs × rt	$op^6 = 0$	rs^5	rt^5	0	0	0x18
multu rs, rt	hi, lo = rs × rt	$op^6 = 0$	rs^5	rt^5	0	0	0x19
div rs, rt	hi, lo = rs / rt	$op^6 = 0$	rs^5	rt^5	0	0	0x1a
divu rs, rt	hi, lo = rs / rt	$op^6 = 0$	rs^5	rt^5	0	0	0x1b
mfhi rd	rd = hi	$op^6 = 0$	0	0	rd^5	0	0x10
mflo rd	rd = lo	$op^6 = 0$	0	0	rd^5	0	0x12
mthi rs	hi = rs	$op^6 = 0$	rs^5	0	0	0	0x11
mtlo rs	lo = rs	$op^6 = 0$	rs^5	0	0	0	0x13

- Signed arithmetic: mult, div (rs and rt are signed)
  - LO = 32-bit low-order and HI = 32-bit high-order of multiplication
  - LO = 32-bit quotient and HI = 32-bit remainder of division
- Unsigned arithmetic: multu, divu (rs and rt are unsigned)
- NO arithmetic exception can occur

❑ I-format (I for Immediate)

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$
  - Address: offset added to base address in rs

✓ **Ex**

✓ **addi $s1 , $s2 , 100**

8	18	17	100
001000	....	....	....
6 bits	5 bits	5 bits	16 bits **Constant**

Load and Store Word

- Load Word Instruction (Word = 4 bytes in MIPS)

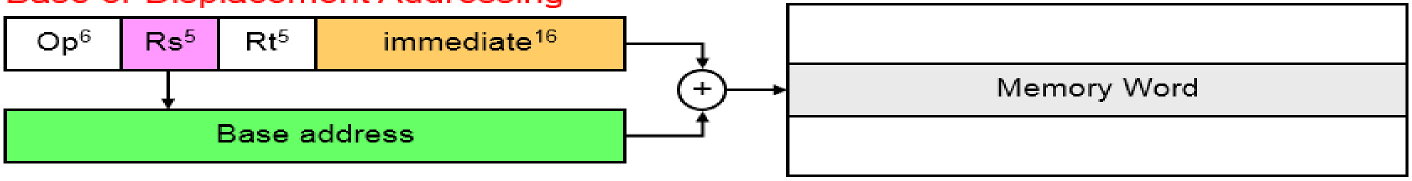
lw Rt, imm^{16}(Rs) # Rt = MEMORY[Rs+imm^{16}]

- Store Word Instruction

sw Rt, imm^{16}(Rs) # MEMORY[Rs+imm^{16}] = Rt

- Base or Displacement addressing is used
  - ✧ Memory Address = Rs (base) + Immediate^{16} (displacement)
  - ✧ Immediate^{16} is sign-extended to have a signed displacement

Base or Displacement Addressing



- ✓ **Ex**
  - ✓ **lw $t0, 1002($s2)**

35	18	8	1002

100011	10010	01000	0000001111101010
6 bits	5 bits	5 bits	16 bits offset

MIPS assembly language

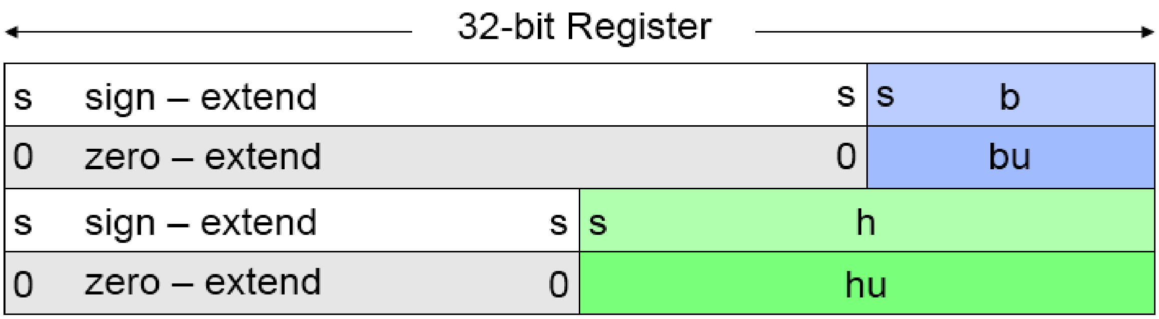
Category	Instruction	Example	Meaning	Comments
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory

MIPS machine language

Name	Format	Example				Comments
lw	I	35	18	17	100	lw $s1,100($s2)
sw	I	43	18	17	100	sw $s1,100($s2)

Load and Store Byte and Halfword

- The MIPS processor supports the following data formats:
  - Byte = 8 bits, Half word = 16 bits, Word = 32 bits
- Load & store instructions for bytes and half words
  - lb = load byte, lbu = load byte unsigned, sb = store byte
  - lh = load half, lhu = load half unsigned, sh = store halfword



Load and Store Instructions

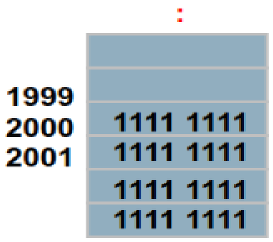
Instruction	Meaning	I-Type Format			
`lb   Rt, imm(Rs)`	Rt $\leftarrow_1$ MEM[Rs+imm]	0x20	Rs	Rt	16-bit immediate
`lh   Rt, imm(Rs)`	Rt $\leftarrow_2$ MEM[Rs+imm]	0x21	Rs	Rt	16-bit immediate
`lw   Rt, imm(Rs)`	Rt $\leftarrow_4$ MEM[Rs+imm]	0x23	Rs	Rt	16-bit immediate
`lbu  Rt, imm(Rs)`	Rt $\leftarrow_1$ MEM[Rs+imm]	0x24	Rs	Rt	16-bit immediate
`lhu  Rt, imm(Rs)`	Rt $\leftarrow_2$ MEM[Rs+imm]	0x25	Rs	Rt	16-bit immediate
`sb   Rt, imm(Rs)`	Rt $\rightarrow_1$ MEM[Rs+imm]	0x28	Rs	Rt	16-bit immediate
`sh   Rt, imm(Rs)`	Rt $\rightarrow_2$ MEM[Rs+imm]	0x29	Rs	Rt	16-bit immediate
`sw   Rt, imm(Rs)`	Rt $\rightarrow_4$ MEM[Rs+imm]	0x2b	Rs	Rt	16-bit immediate

- **Base / Displacement Addressing** is used
  - Memory Address = Rs (**Base**) + Immediate (**displacement**)
  - If Rs is $zero then   Address = Immediate (**absolute**)
  - If Immediate is 0 then   Address = Rs (**register indirect**)

Example

- We want to load a BYTE into **$s3** from the address 2000

  After the load, what is the value of **$s3** ?



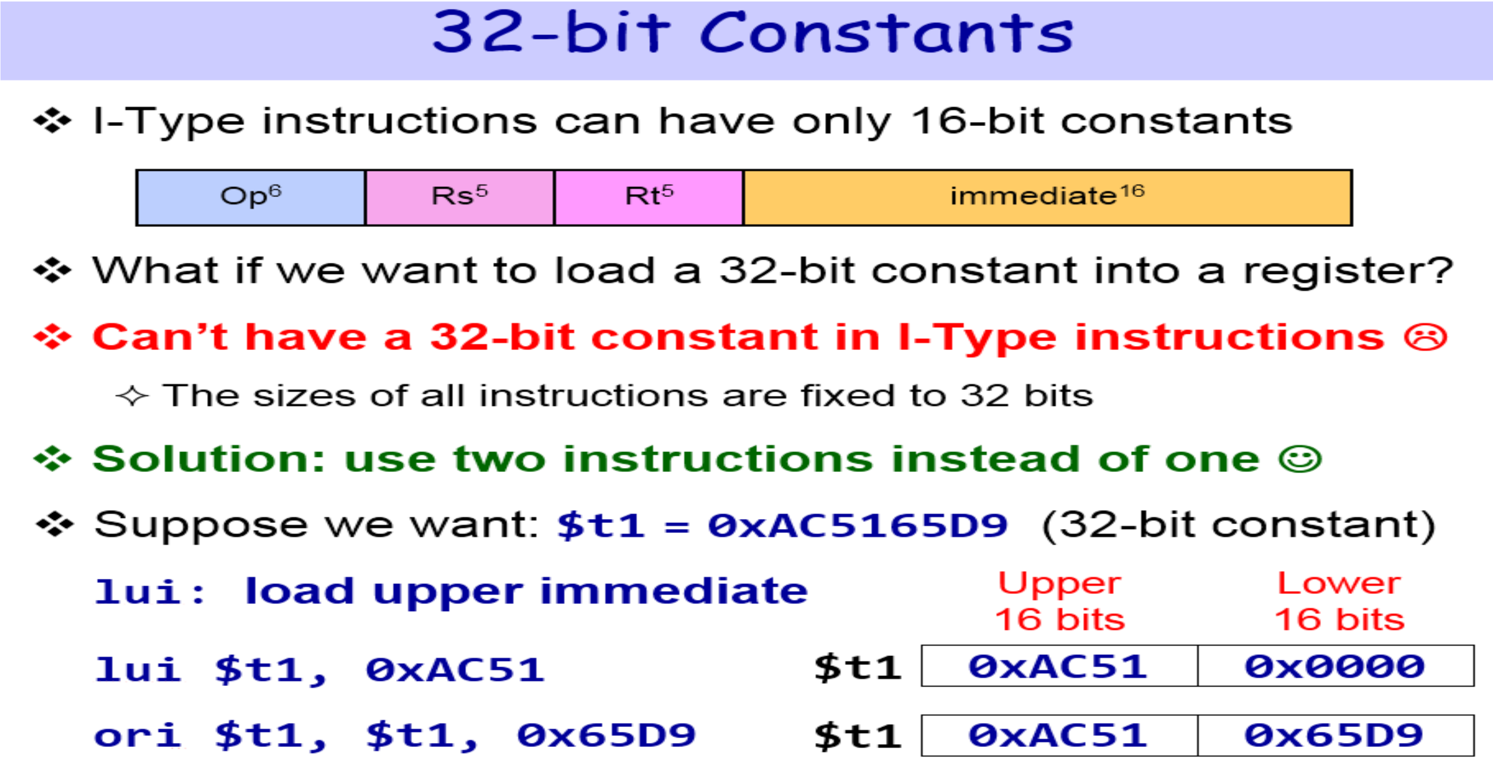
Assume
$s0 = 2000

A1 - Unsigned → `lbu $s3, 0($s0)`

- A1: 0000 0000 0000 0000 0000 0000 1111 1111 (255) ?

A2 - Signed → `lb  $s3, 0($s0)`

- A2: 1111 1111 1111 1111 1111 1111 1111 1111 (–1) ?



32-bit Constants

- I-Type instructions can have only 16-bit constants

Op⁶	Rs⁵	Rt⁵	immediate¹⁶

- What if we want to load a 32-bit constant into a register?
- Can't have a 32-bit constant in I-Type instructions ☹
  - The sizes of all instructions are fixed to 32 bits
- Solution: use two instructions instead of one ☺
- Suppose we want: **$t1 = 0xAC5165D9** (32-bit constant)

lui: load upper immediate

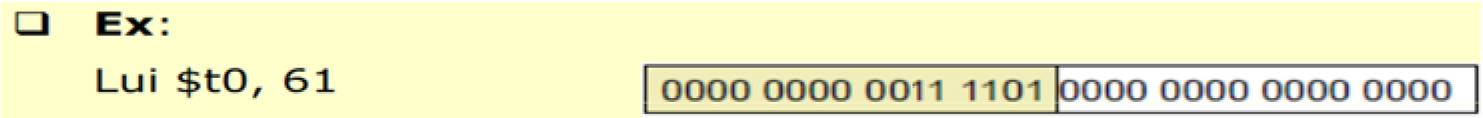
		Upper 16 bits	Lower 16 bits
`lui $t1, 0xAC51`	$t1	0xAC51	0x0000
`ori $t1, $t1, 0x65D9`	$t1	0xAC51	0x65D9

32-Bit Immediate Operands

Most constants are small 16-bit immediate is sufficient. The MIPS instruction set includes the instruction load upper immediate (lui).

lui (load upper immediate):

Transfers the 16-bit immediate constant field value into the left most 16 bits of the register (upper half word of register), filling the lower 16 bits with 0s.

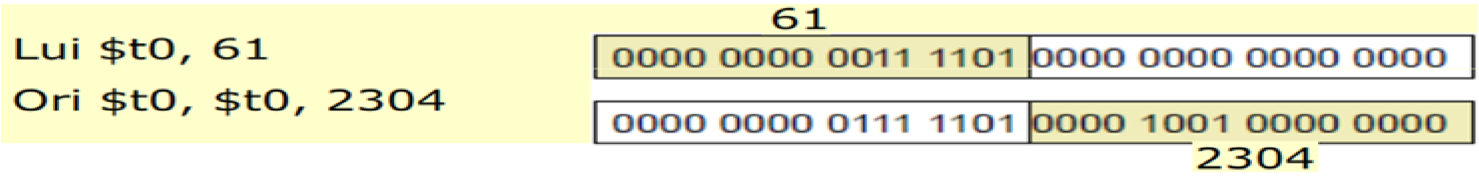


❑ **Ex:**

Lui $t0, 61	0000 0000 0011 1101	0000 0000 0000 0000

Ex: What is the MIPS assembly code to load this 32-bit constant into register $s0?

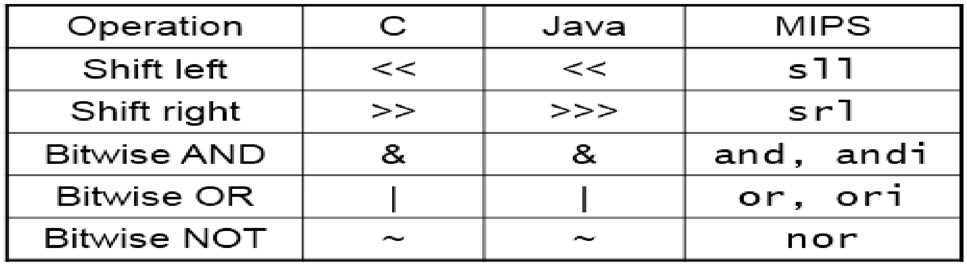
0000 0000 0011 1101 0000 1001 0000 0000



	61	
Lui $t0, 61	0000 0000 0011 1101	0000 0000 0000 0000
Ori $t0, $t0, 2304	0000 0000 0111 1101	0000 1001 0000 0000
		2304

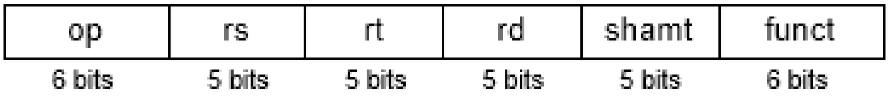
Logical Operations

(Instructions for bitwise manipulation)



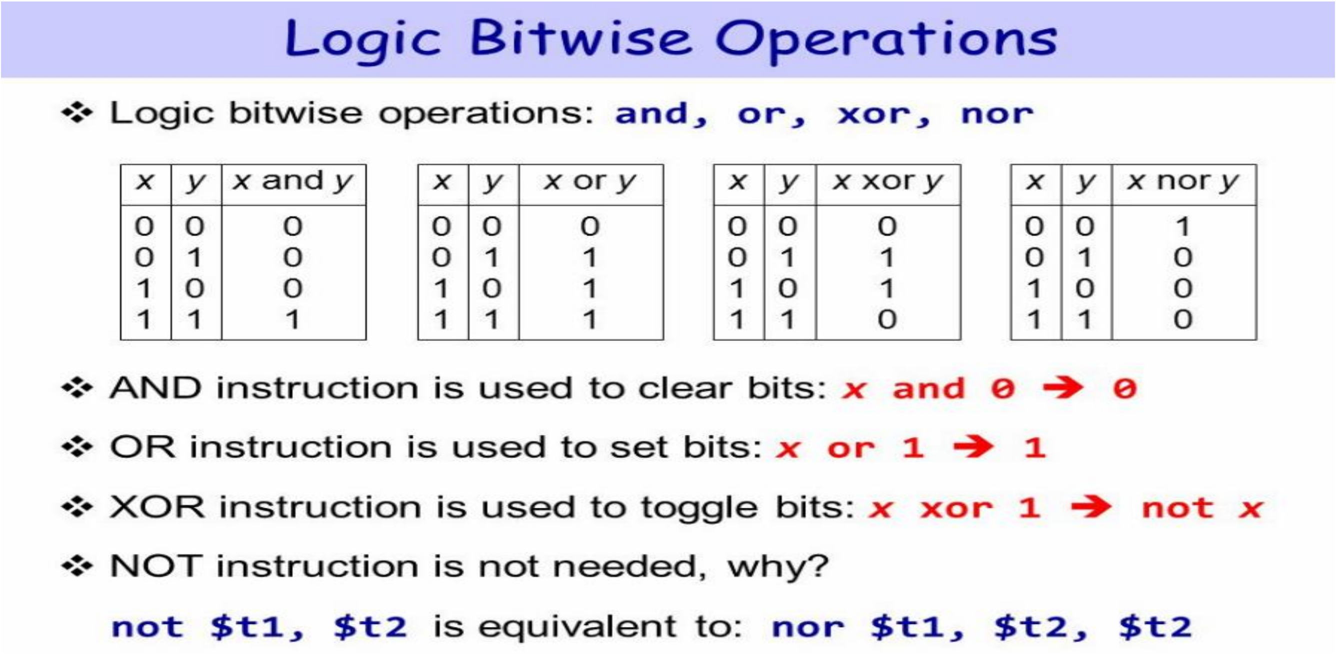
Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- **Useful for extracting and inserting groups of bits in a word**
- **Shift Operations**
  - ***Shift left logical***
    - Shift left and fill with 0 bits
    - **sll** by ***i*** bits multiplies by $\mathbf{2}^{i}$
  - ***Shift right logical***
    - Shift right and fill with 0 bits
    - **srl** by ***i*** bits divides by $\mathbf{2}^{i}$ (unsigned only)
  - ***Shift instruction format (R- format)***



op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

shamt: how many positions to shift



Logic Bitwise Operations

- Logic bitwise operations: **and, or, xor, nor**

x	y	x and y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x or y
0	0	0
0	1	1
1	0	1
1	1	1

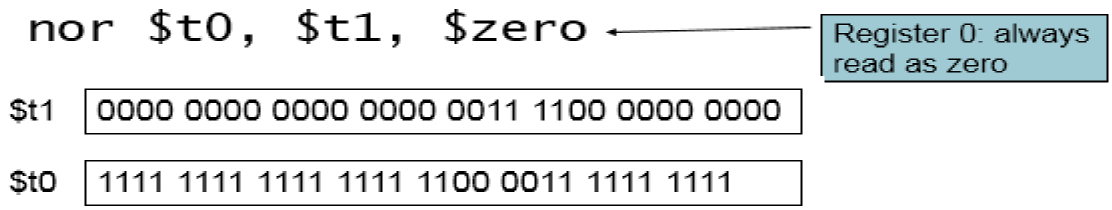
x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0

x	y	x nor y
0	0	1
0	1	0
1	0	0
1	1	0

- AND instruction is used to clear bits: **x and 0 → 0**
- OR instruction is used to set bits: **x or 1 → 1**
- XOR instruction is used to toggle bits: **x xor 1 → not x**
- NOT instruction is not needed, why?

not $t1, $t2 is equivalent to: **nor $t1, $t2, $t2**

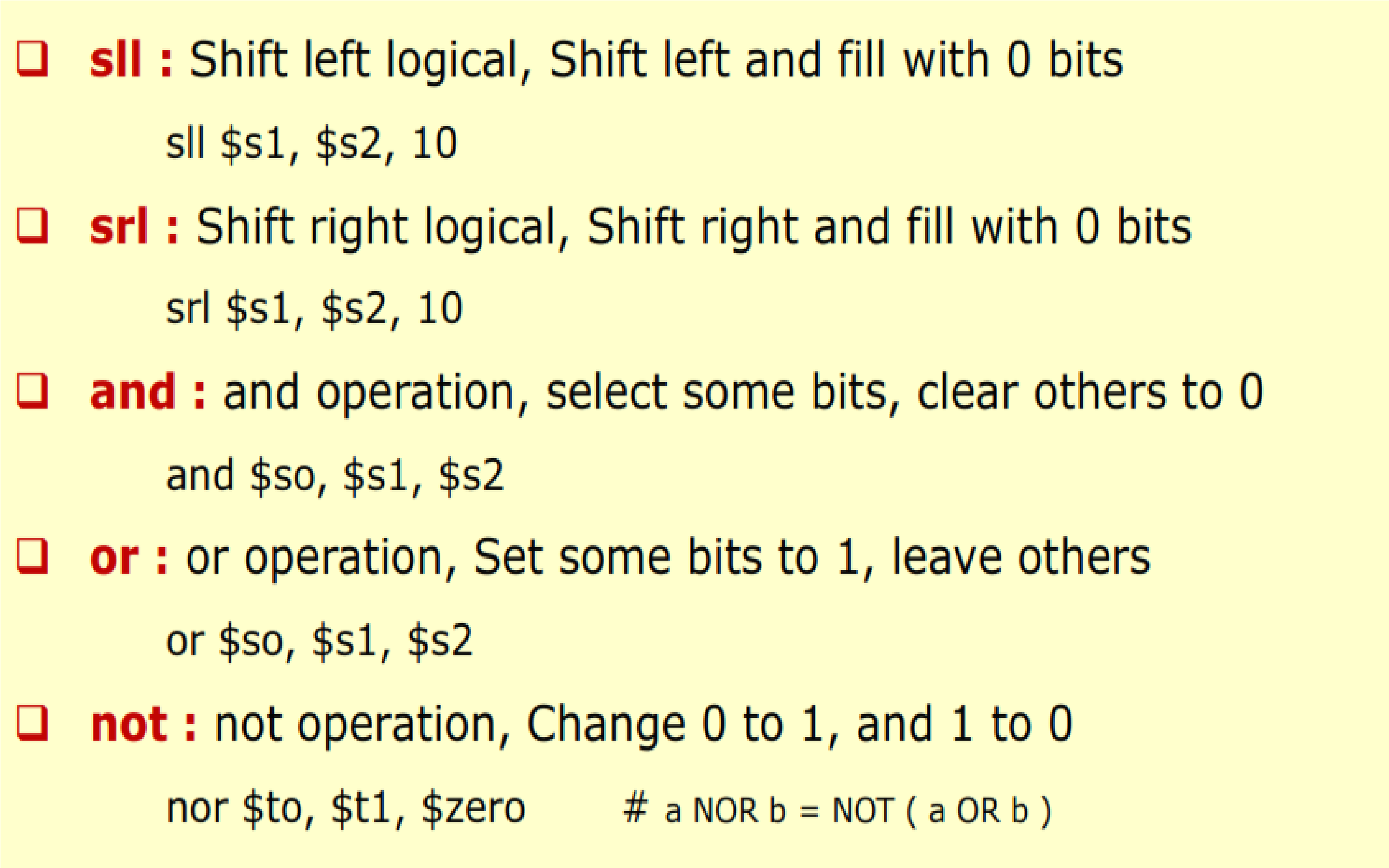
- **AND Operations**
  - Useful to mask bits in a word (clear some bits to 0)
- **OR Operations**
  - Useful to include bits in a word,(set some bits to 1)
- **NOT Operations**
  - Useful to invert bits in a word Change 0 to 1, and 1 to 0
  - MIPS has NOR 3-operand instruction
    - a NOR b == NOT (a OR b)



```
nor $t0, $t1, $zero
```

Register 0: always read as zero

```
$t1   0000 0000 0000 0000 0011 1100 0000 0000
$t0   1111 1111 1111 1111 1100 0011 1111 1111
```



❑ **sll :** Shift left logical, Shift left and fill with 0 bits

```
sll $s1, $s2, 10
```

❑ **srl :** Shift right logical, Shift right and fill with 0 bits

```
srl $s1, $s2, 10
```

❑ **and :** and operation, select some bits, clear others to 0

```
and $so, $s1, $s2
```

❑ **or :** or operation, Set some bits to 1, leave others

```
or $so, $s1, $s2
```

❑ **not :** not operation, Change 0 to 1, and 1 to 0

```
nor $to, $t1, $zero      # a NOR b = NOT ( a OR b )
```

Logical Bitwise Instructions

Instruction	Meaning	R-Type Format					
and $s1, $s2, $s3	$s1 = $s2 & $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x24
or $s1, $s2, $s3	$s1 = $s2 | $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x25
xor $s1, $s2, $s3	$s1 = $s2 ^ $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x26
nor $s1, $s2, $s3	$s1 = ~($s2|$s3)	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x27

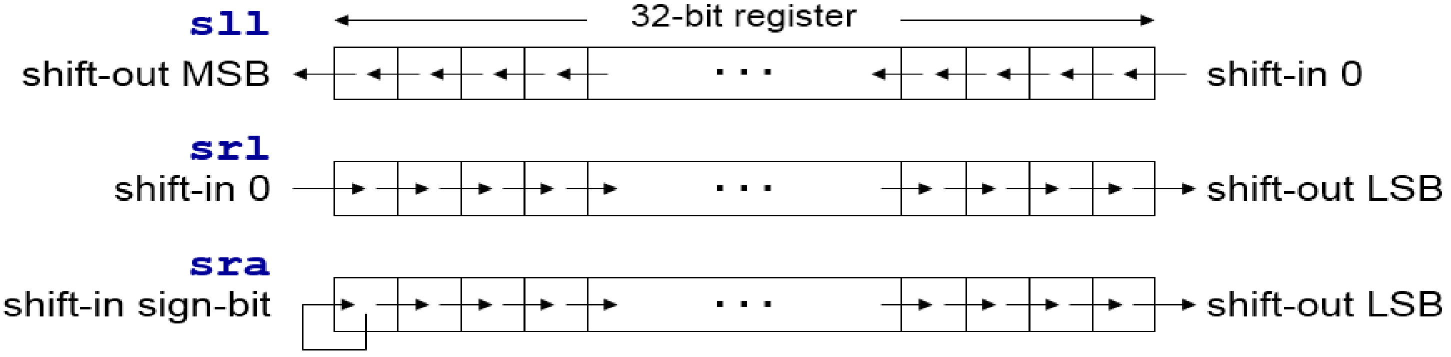
- Examples:

Assume `$s1 = 0xabcd1234` and `$s2 = 0xffff0000`

```
and $s0,$s1,$s2     # $s0 = 0xabcd0000
or  $s0,$s1,$s2     # $s0 = 0xffff1234
xor $s0,$s1,$s2     # $s0 = 0x54321234
nor $s0,$s1,$s2     # $s0 = 0x0000edcb
```

Shift Operations

- Shifting is to move all the bits in a register left or right
- Shifts by a constant amount: `sll, srl, sra`
  - `sll/srl` mean shift left/right logical by a constant amount
  - The 5-bit shift amount field is used by these instructions
  - `sra` means shift right arithmetic by a constant amount
  - The sign-bit (rather than 0) is shifted from the left



Shift Instructions

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
sll $t1,$t2,10	$t1 = $t2 << 10	0	0	$t2	$t1	10	0
srl $t1,$t2,10	$t1 = $t2 >>> 10	0	0	$t2	$t1	10	2
sra $t1,$t2,10	$t1 = $t2 >> 10	0	0	$t2	$t1	10	3
sllv $t1,$t2,$t3	$t1 = $t2 << $t3	0	$t3	$t2	$t1	0	4
srlv $t1,$t2,$t3	$t1 = $t2 >>>$t3	0	$t3	$t2	$t1	0	6
srav $t1,$t2,$t3	$t1 = $t2 >> $t3	0	$t3	$t2	$t1	0	7

- **sll, srl, sra: shift by a constant amount**
  - The shift amount (sa) field specifies a number between 0 and 31
- **sllv, srlv, srav: shift by a variable amount**
  - A source register specifies the variable shift amount between 0 and 31

Examples

- Given that: **$t2 = 0xabcd1234 and $t3 = 16**

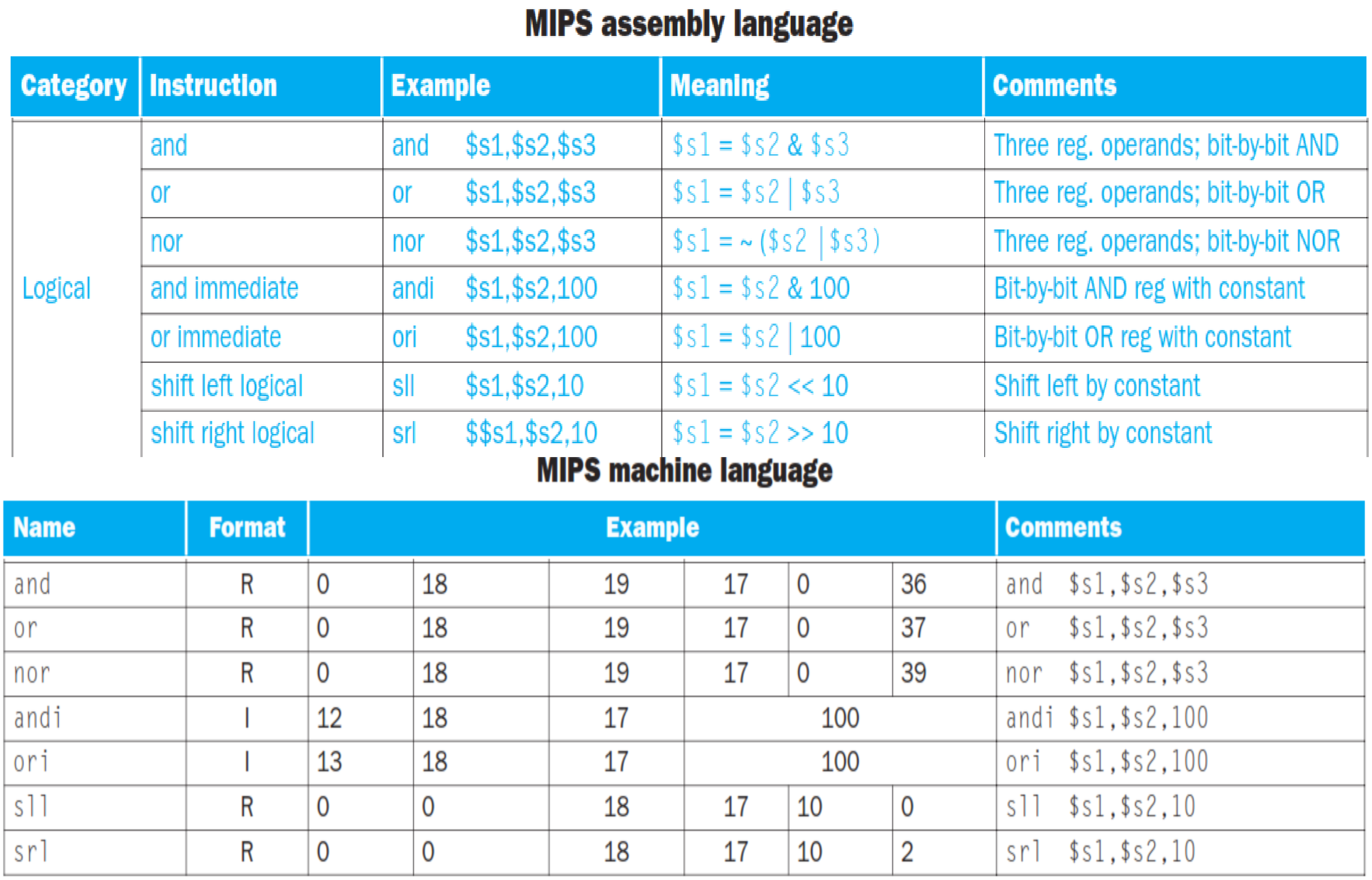
sll $t1, $t2, 8	$t1 = 0xcd123400
srl $t1, $t2, 4	$t1 = 0x0abcd123
sra $t1, $t2, 4	$t1 = 0xfabcd123
srlv $t1, $t2, $t3	$t1 = 0x0000abcd

Op	Rs = $t3	Rt = $t2	Rd = $t1	sa	srlv
000000	01011	01010	01001	00000	000110

- ✧ Example: multiply **$s1** by **36**
  - Factor 36 into (4 + 32) and use distributive property of multiplication
- ✧ **$s2 = $s1*36 = $s1*(4 + 32) = $s1*4 + $s1*32**

```
sll  $t0, $s1, 2       ; $t0 = $s1 * 4
sll  $t1, $s1, 5       ; $t1 = $s1 * 32
addu $s2, $t0, $t1     ; $s2 = $s1 * 36
```

Note: Logical AND immediate and logical OR immediate put 0s into the upper 16 bits to form a 32-bit constant, unlike add immediate, which does sign extension.

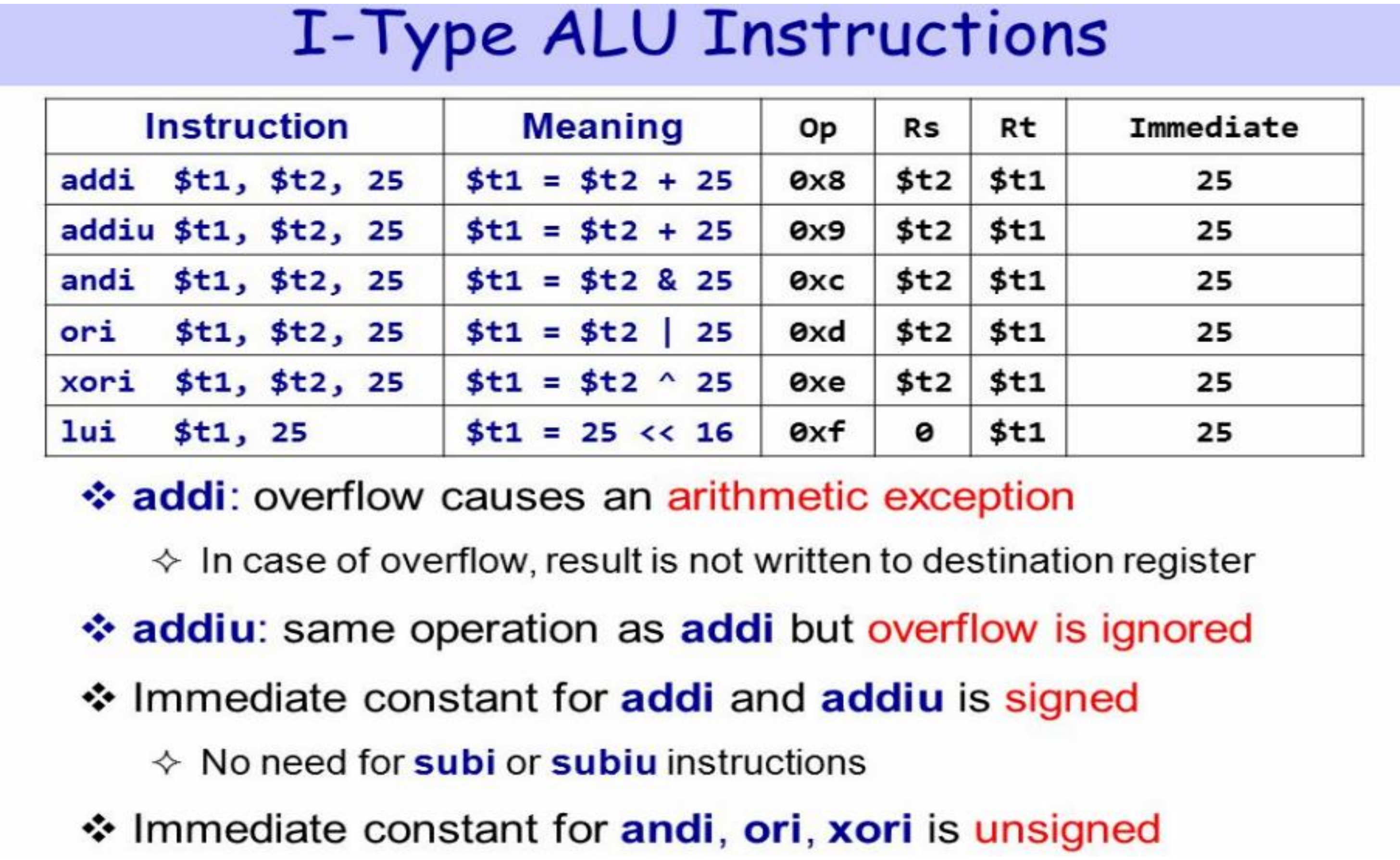


MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant

MIPS machine language

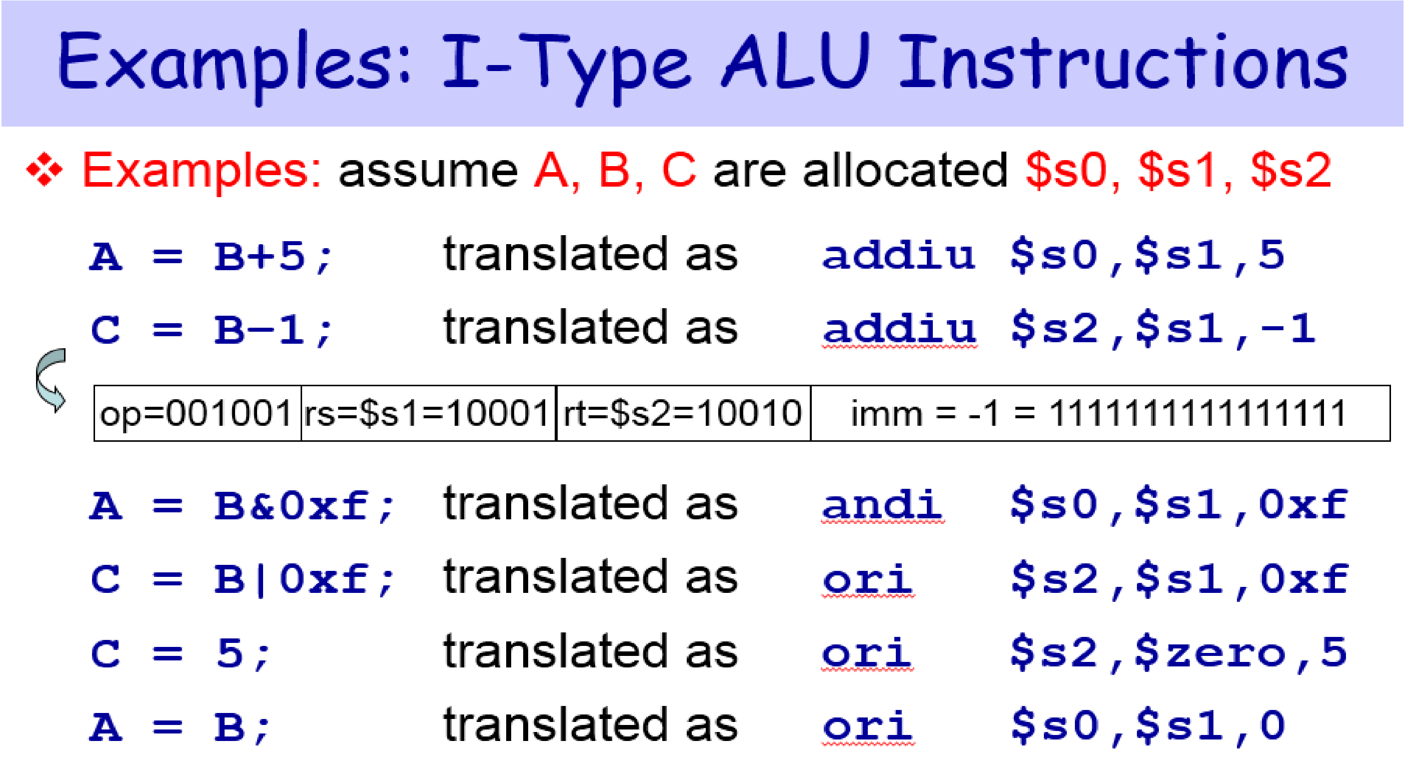
Name	Format	Example						Comments
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10



I-Type ALU Instructions

Instruction	Meaning	Op	Rs	Rt	Immediate
`addi  $t1, $t2, 25`	`$t1 = $t2 + 25`	0x8	$t2	$t1	25
`addiu $t1, $t2, 25`	`$t1 = $t2 + 25`	0x9	$t2	$t1	25
`andi  $t1, $t2, 25`	`$t1 = $t2 & 25`	0xc	$t2	$t1	25
`ori   $t1, $t2, 25`	`$t1 = $t2 | 25`	0xd	$t2	$t1	25
`xori  $t1, $t2, 25`	`$t1 = $t2 ^ 25`	0xe	$t2	$t1	25
`lui   $t1, 25`	`$t1 = 25 << 16`	0xf	0	$t1	25

- **addi**: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register
- **addiu**: same operation as **addi** but overflow is ignored
- Immediate constant for **addi** and **addiu** is signed
  - No need for **subi** or **subiu** instructions
- Immediate constant for **andi**, **ori**, **xori** is unsigned



Examples: I-Type ALU Instructions

- Examples: assume A, B, C are allocated $s0, $s1, $s2

`A = B+5;` translated as `addiu $s0,$s1,5`

`C = B–1;` translated as `addiu $s2,$s1,-1`

op=001001	rs=$s1=10001	rt=$s2=10010	imm = -1 = 1111111111111111

`A = B&0xf;` translated as `andi $s0,$s1,0xf`

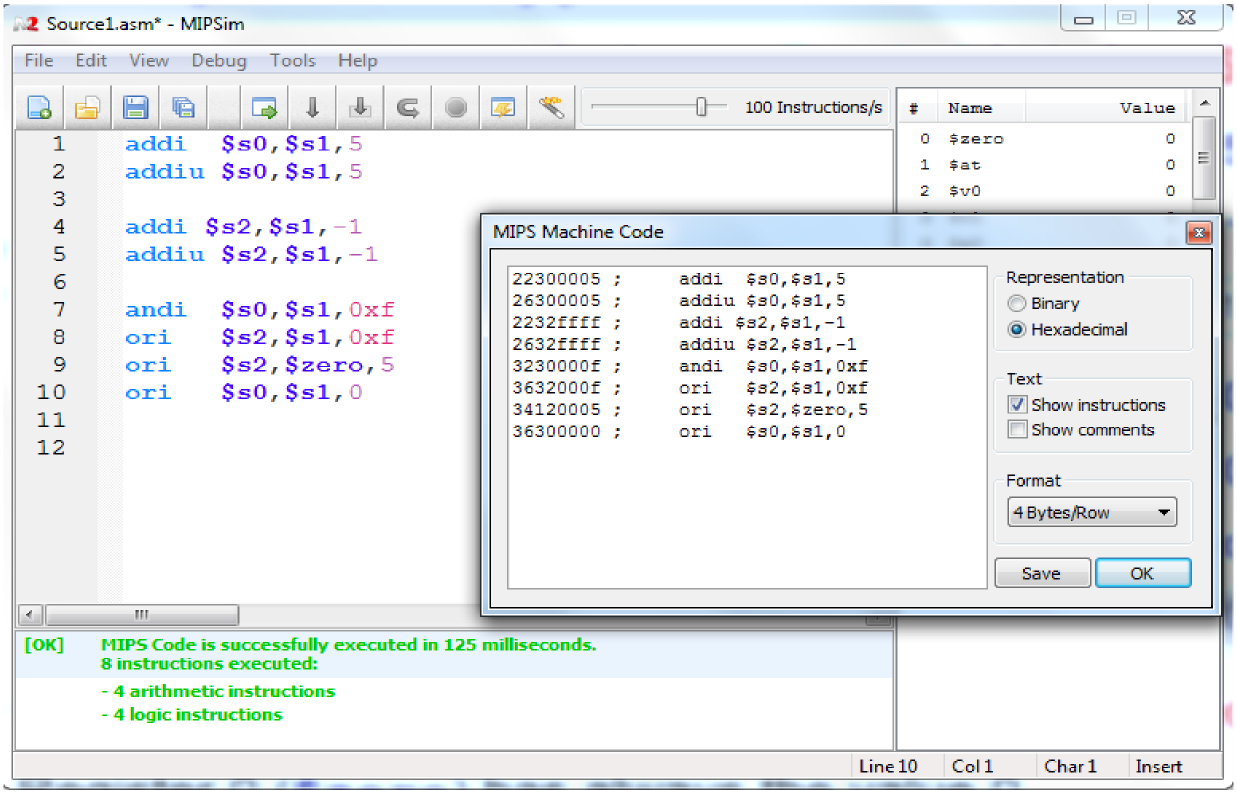
`C = B|0xf;` translated as `ori $s2,$s1,0xf`

`C = 5;` translated as `ori $s2,$zero,5`

`A = B;` translated as `ori $s0,$s1,0`

http://www.mipsim.tk/download.aspx

(MIPS) Microprocessor without Interlocked Pipeline Stages



```
addi   $s0,$s1,5
addiu  $s0,$s1,5

addi  $s2,$s1,-1
addiu  $s2,$s1,-1

andi   $s0,$s1,0xf
ori    $s2,$s1,0xf
ori    $s2,$zero,5
ori    $s0,$s1,0
```

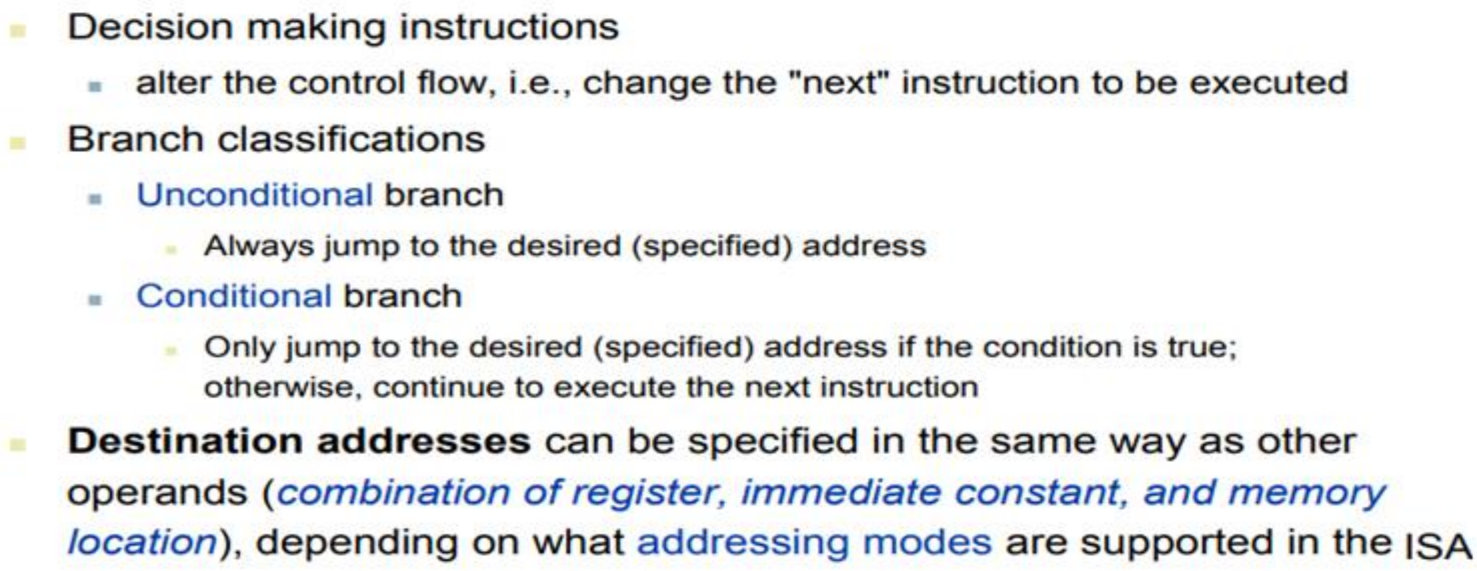
#	Name	Value
0	$zero	0
1	$at	0
2	$v0	0

```
22300005 ;     addi  $s0,$s1,5
26300005 ;     addiu $s0,$s1,5
2232ffff ;     addi  $s2,$s1,-1
2632ffff ;     addiu $s2,$s1,-1
3230000f ;     andi  $s0,$s1,0xf
3632000f ;     ori   $s2,$s1,0xf
34120005 ;     ori   $s2,$zero,5
36300000 ;     ori   $s0,$s1,0
```

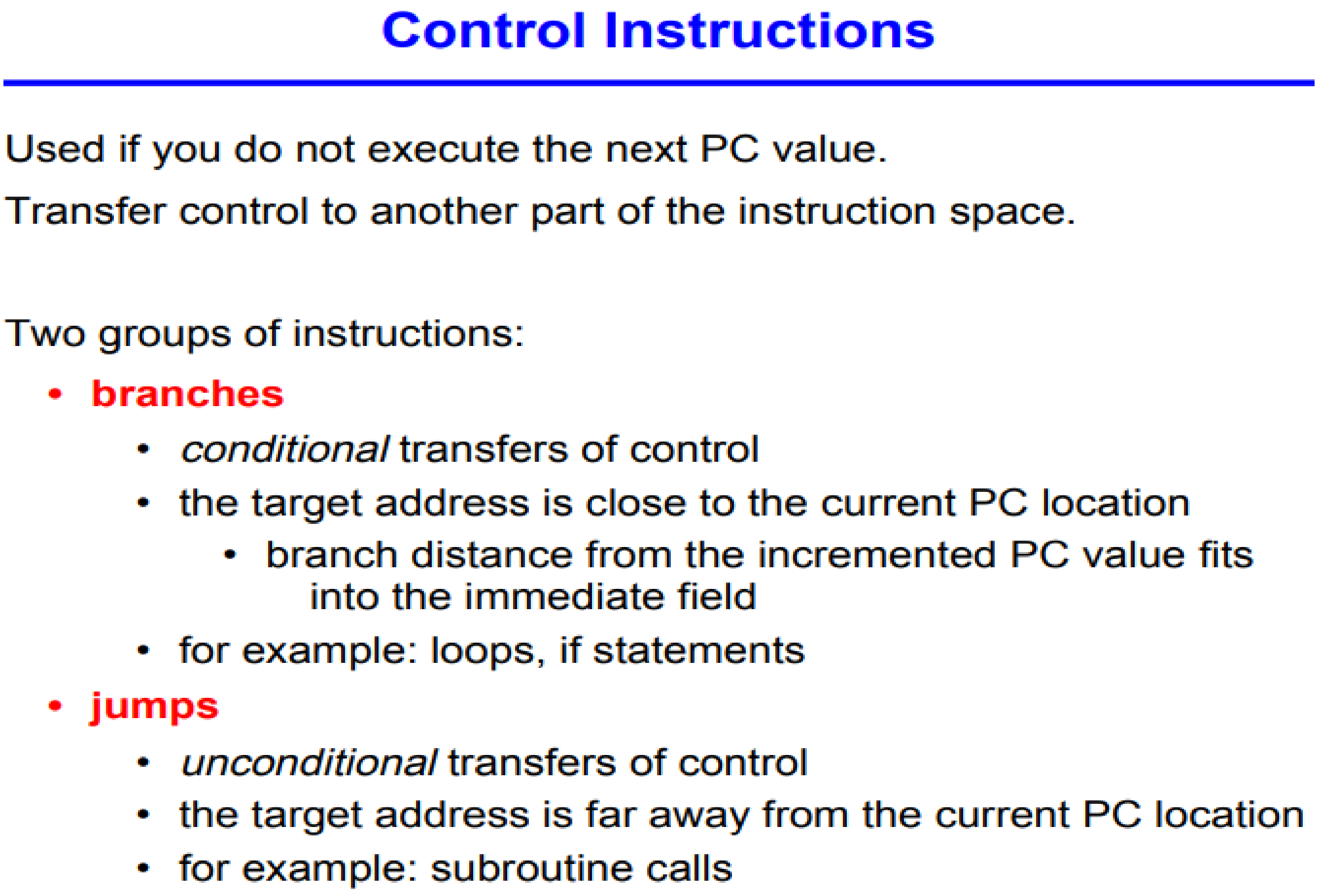
```
[OK]    MIPS Code is successfully executed in 125 milliseconds.
        8 instructions executed:

        - 4 arithmetic instructions
        - 4 logic instructions
```

Control Flow

Branch and Jump Instructions



- Decision making instructions
  - alter the control flow, i.e., change the "next" instruction to be executed
- Branch classifications
  - Unconditional branch
    - Always jump to the desired (specified) address
  - Conditional branch
    - Only jump to the desired (specified) address if the condition is true; otherwise, continue to execute the next instruction
- **Destination addresses** can be specified in the same way as other operands (*combination of register, immediate constant, and memory location*), depending on what addressing modes are supported in the ISA



Control Instructions

Used if you do not execute the next PC value.

Transfer control to another part of the instruction space.

Two groups of instructions:

- **branches**
  - *conditional* transfers of control
  - the target address is close to the current PC location
    - branch distance from the incremented PC value fits into the immediate field
  - for example: loops, if statements
- **jumps**
  - *unconditional* transfers of control
  - the target address is far away from the current PC location
  - for example: subroutine calls

❖ The Jump instruction is of the J-type format:

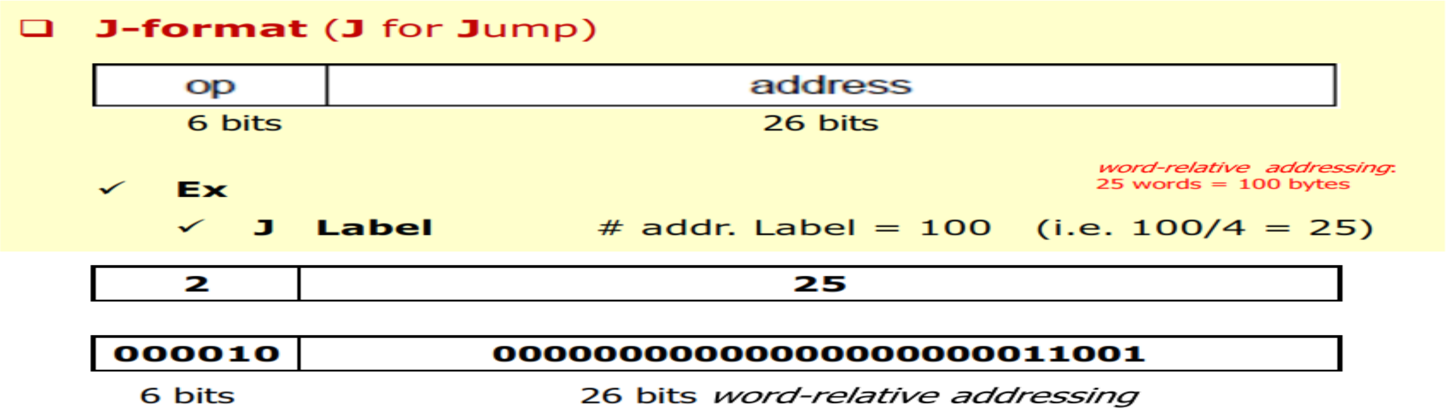
$Op^6 = 2$	26-bit address

❖ The jump instruction modifies the program counter PC:

PC^4	26-bit address	00

multiple of 4

❖ The upper 4 bits of the PC are unchanged



❑ **J-format (J for Jump)**

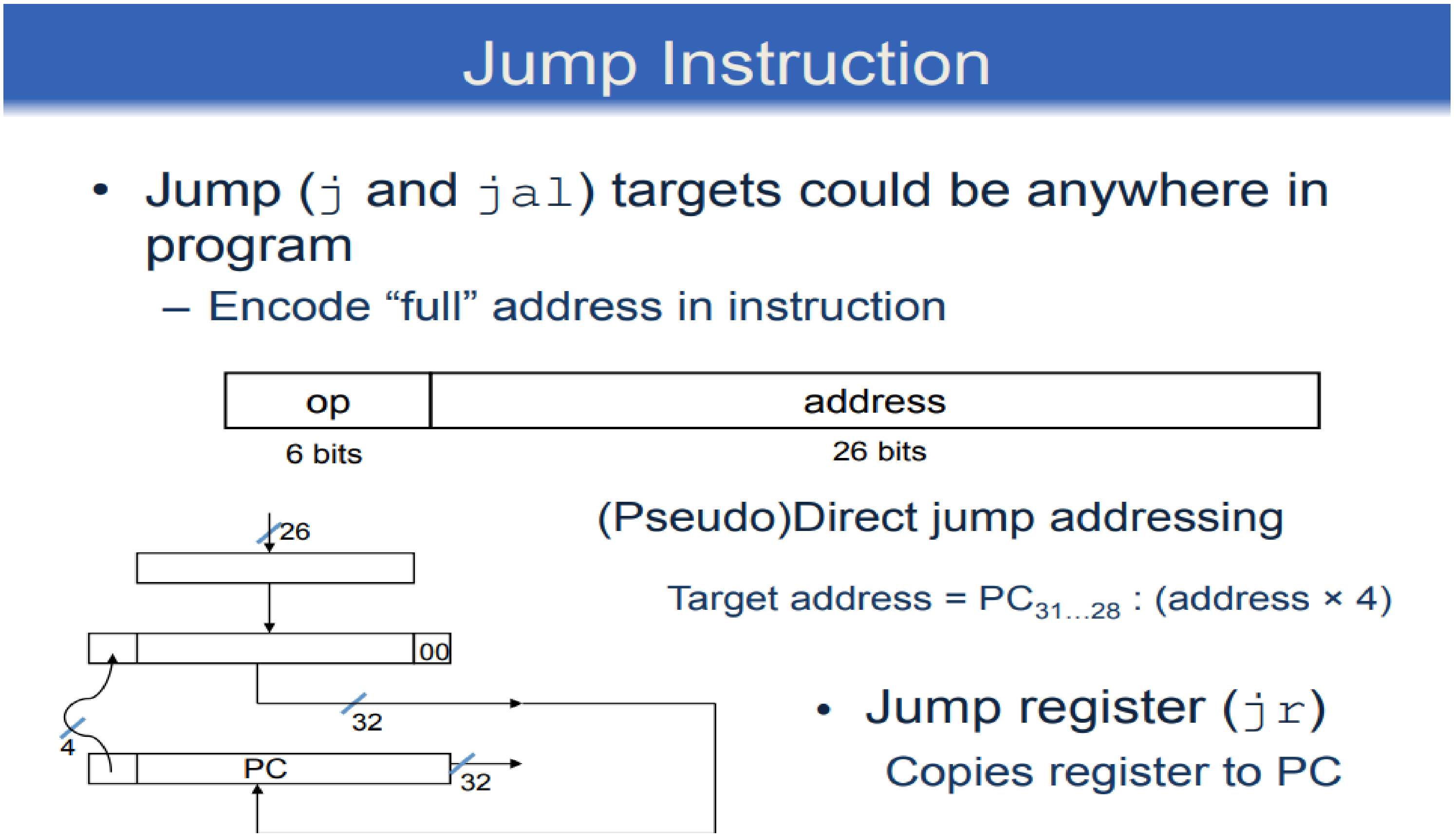
op	address
6 bits	26 bits

✓ **Ex**

word-relative addressing:
25 words = 100 bytes

✓ **J Label** # addr. Label = 100 (i.e. 100/4 = 25)

2	**25**
000010	**00000000000000000000011001**
6 bits	26 bits *word-relative addressing*



Jump Instruction

- Jump (`j` and `jal`) targets could be anywhere in program
  - Encode "full" address in instruction

op	address
6 bits	26 bits

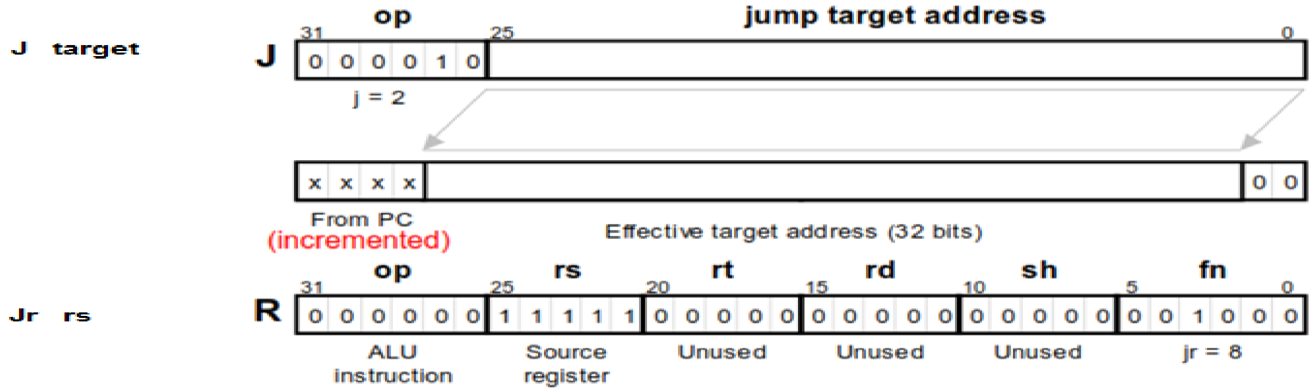
26
00
4
32
PC
32

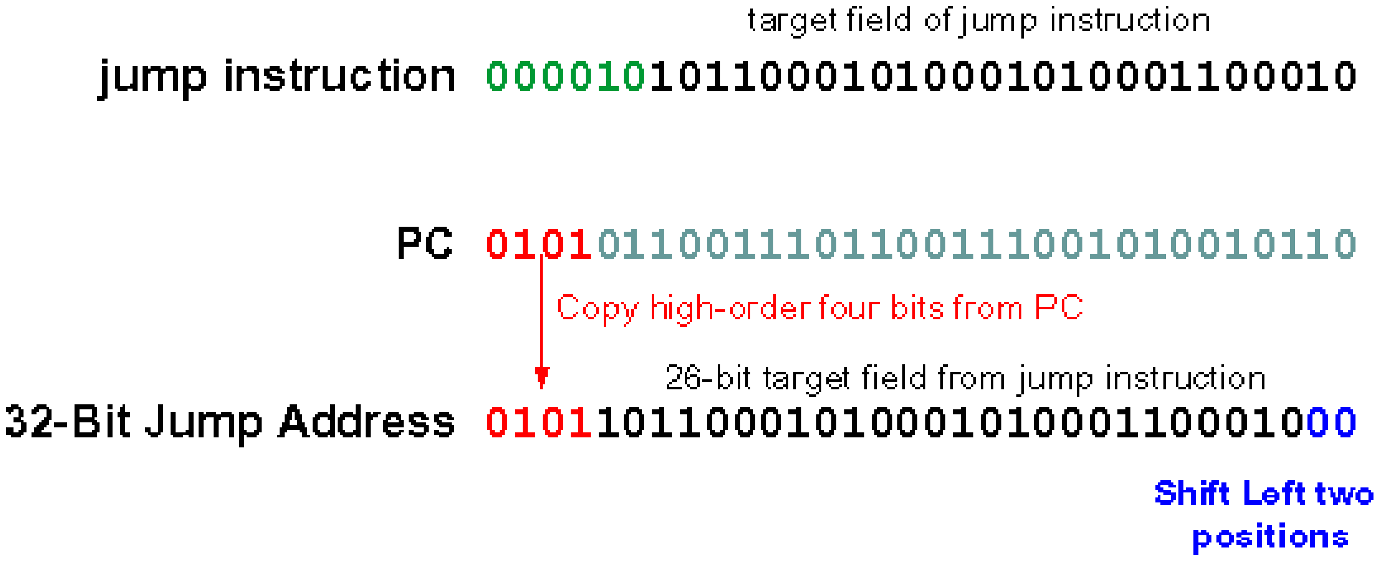
(Pseudo)Direct jump addressing

Target address = $PC_{31\ldots28}$: (address × 4)

- Jump register (`jr`)

  Copies register to PC





target field of jump instruction

jump instruction 00001010110001010001010001100010

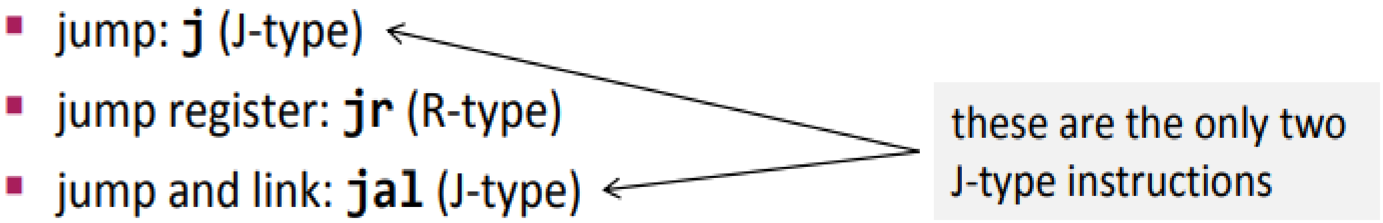
PC 01010110011101100111001010010110

Copy high-order four bits from PC

26-bit target field from jump instruction

32-Bit Jump Address 01011011000101000101000110001000

Shift Left two positions



- jump: **j** (J-type)
- jump register: **jr** (R-type)
- jump and link: **jal** (J-type)

these are the only two J-type instructions

Instruction		Opcode	Target
j	label	000010	coded address of label
jal	label	000011	coded address of label

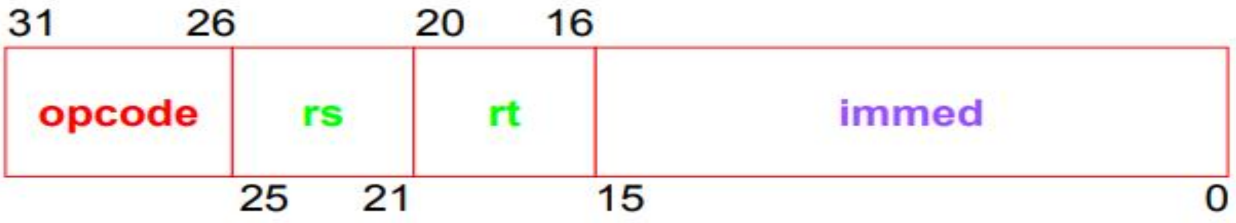
MIPS Jump Instructions

Jump instructions: unconditional transfer of control

j	target	# jump go to the specified target address
jr	rs	# jump register go to the address stored in rs (called an **indirect jump**)
jal	target	# jump and link go to the target address; save PC+4 in $ra
jalr	rs, rd	# jump and link register go to the address stored in rs; rd = PC+4 default rd is $ra

I-type Format for Branches

I-type format used for conditional branches



- **opcode** = control instruction
- **rs, rt** = source operands
- immed = address offset in words, $\pm 2^{15}$
  - hardware sign-extends when uses (replicate msb)
  - target address = PC + (immed*4)

MIPS Conditional Branch Instructions

- MIPS **compare and branch** instructions:

  `beq Rs, Rt, label` if (`Rs == Rt`) branch to `label`

  `bne Rs, Rt, label` if (`Rs != Rt`) branch to `label`

- MIPS **compare to zero & branch** instructions:

  Compare to zero is used frequently and implemented efficiently

  `bltz Rs, label` if (`Rs < 0`) branch to `label`

  `bgtz Rs, label` if (`Rs > 0`) branch to `label`

  `blez Rs, label` if (`Rs <= 0`) branch to `label`

  `bgez Rs, label` if (`Rs >= 0`) branch to `label`

- `beqz` and `bnez` are defined as pseudo-instructions.

Branch Instruction Format

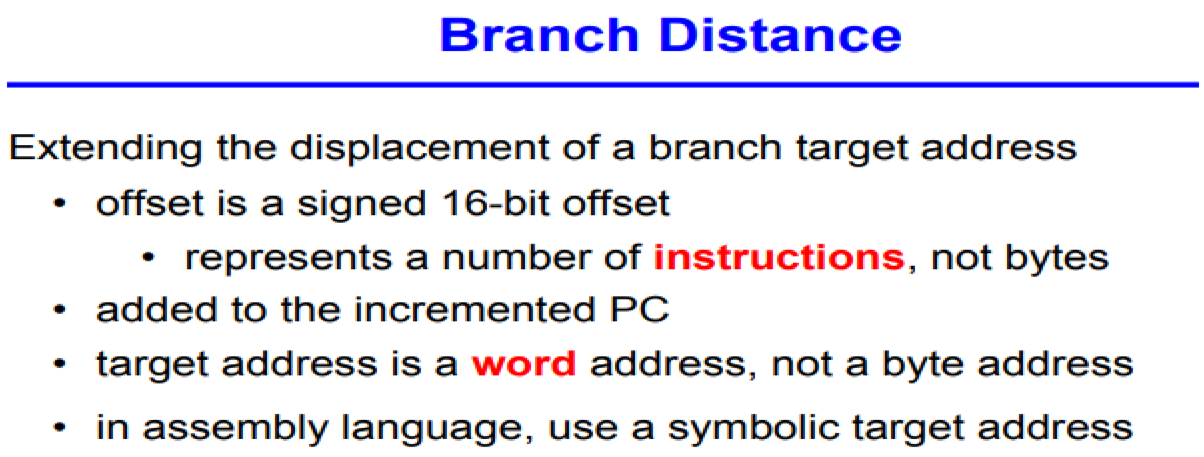
- Branch Instructions are of the I-type Format:

Op^6	Rs^5	Rt^5	16-bit offset

Instruction	I-Type Format			
`beq  Rs, Rt, label`	Op = 4	Rs	Rt	16-bit Offset
`bne  Rs, Rt, label`	Op = 5	Rs	Rt	16-bit Offset
`blez Rs, label`	Op = 6	Rs	0	16-bit Offset
`bgtz Rs, label`	Op = 7	Rs	0	16-bit Offset
`bltz Rs, label`	Op = 1	Rs	0	16-bit Offset
`bgez Rs, label`	Op = 1	Rs	1	16-bit Offset

- The branch instructions modify the **PC register** only
- **PC-Relative addressing**:

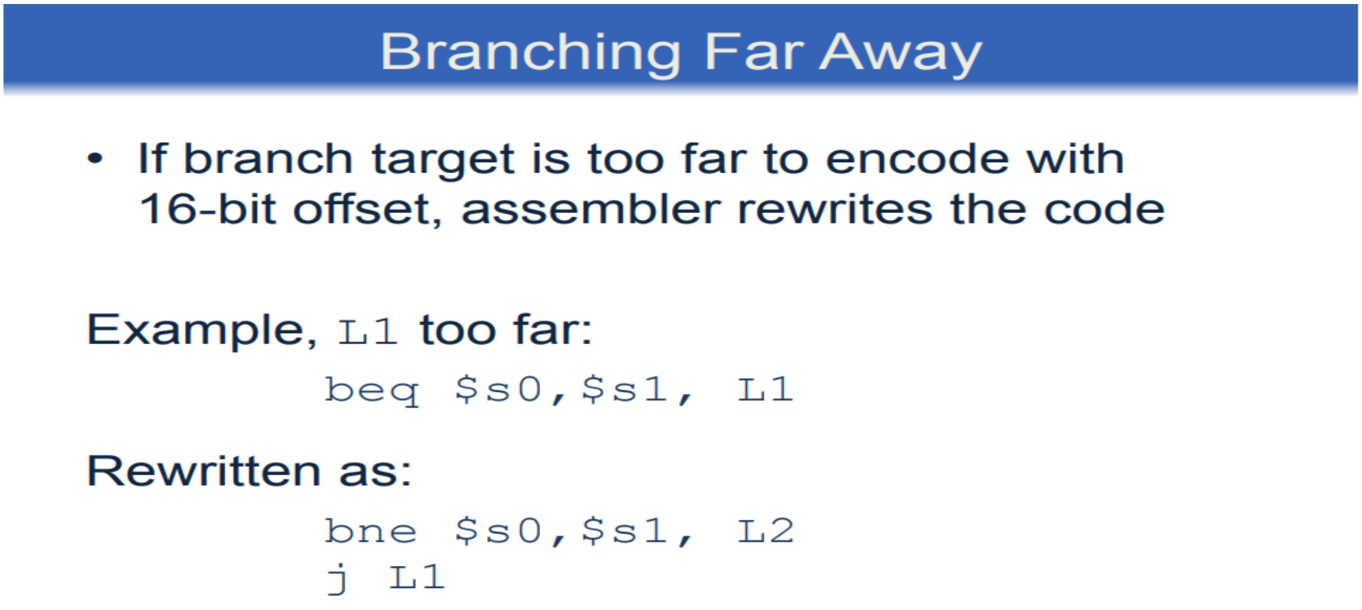
  If (branch is taken) **PC = PC + 4 + 4×offset** else **PC = PC+4**



Branch Distance

Extending the displacement of a branch target address

- offset is a signed 16-bit offset
    - represents a number of **instructions**, not bytes
- added to the incremented PC
- target address is a **word** address, not a byte address
- in assembly language, use a symbolic target address



Branching Far Away

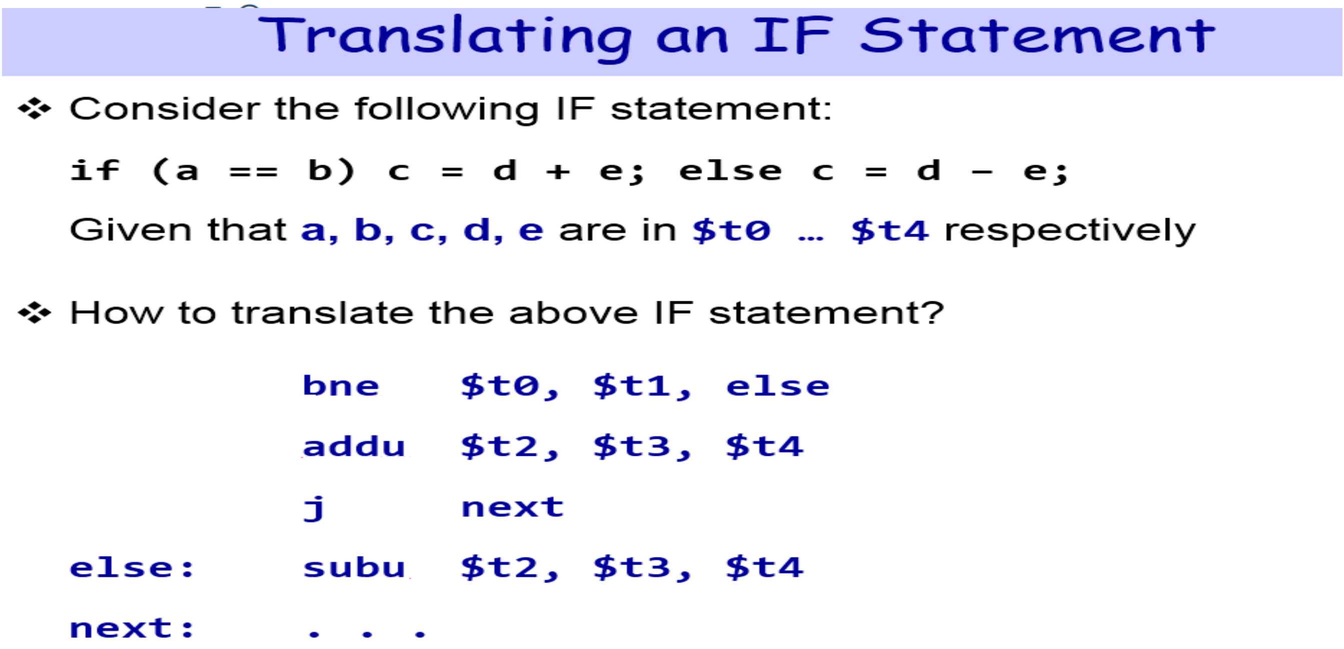
- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

Example, `L1` too far:

```
beq $s0,$s1, L1
```

Rewritten as:

```
bne $s0,$s1, L2
j L1
```



Translating an IF Statement

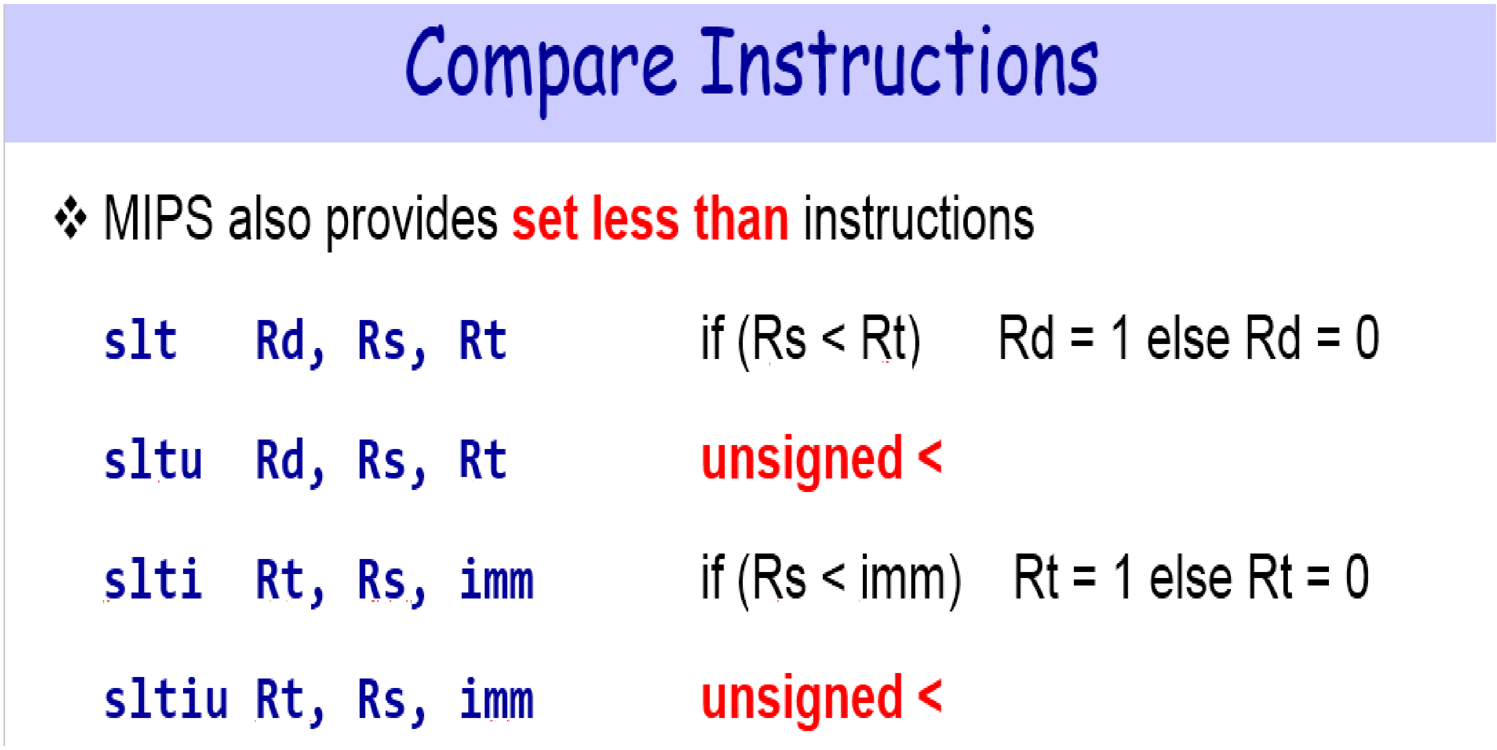
❖ Consider the following IF statement:

```
if (a == b) c = d + e; else c = d – e;
```

Given that **a, b, c, d, e** are in **$t0 … $t4** respectively

❖ How to translate the above IF statement?

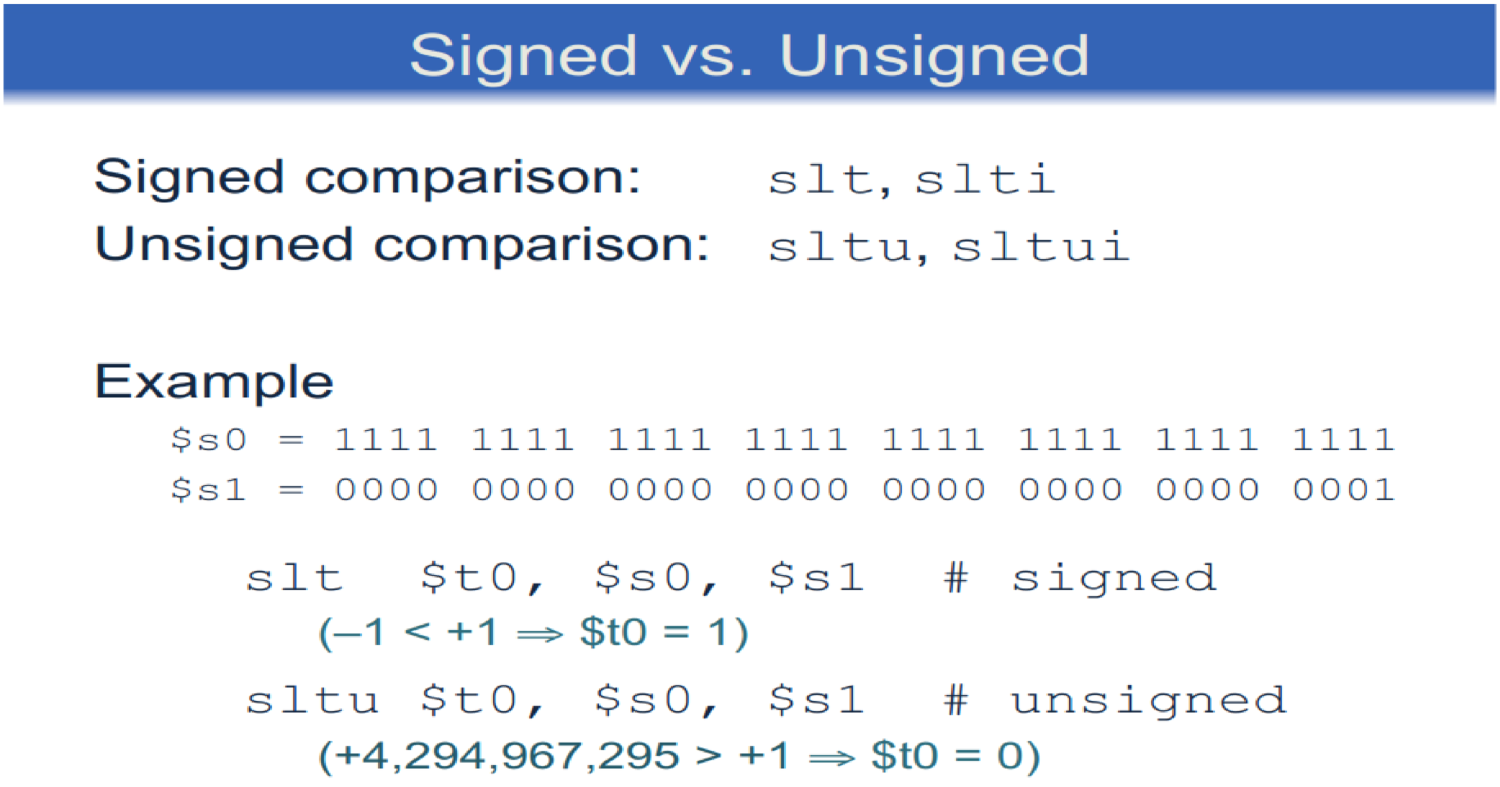
```
         bne    $t0, $t1, else
         addu   $t2, $t3, $t4
         j      next
else:    subu   $t2, $t3, $t4
next:    . . .
```



Compare Instructions

- ❖ MIPS also provides **set less than** instructions

```
slt   Rd, Rs, Rt      if (Rs < Rt)   Rd = 1 else Rd = 0
sltu  Rd, Rs, Rt      unsigned <
slti  Rt, Rs, imm     if (Rs < imm)  Rt = 1 else Rt = 0
sltiu Rt, Rs, imm     unsigned <
```

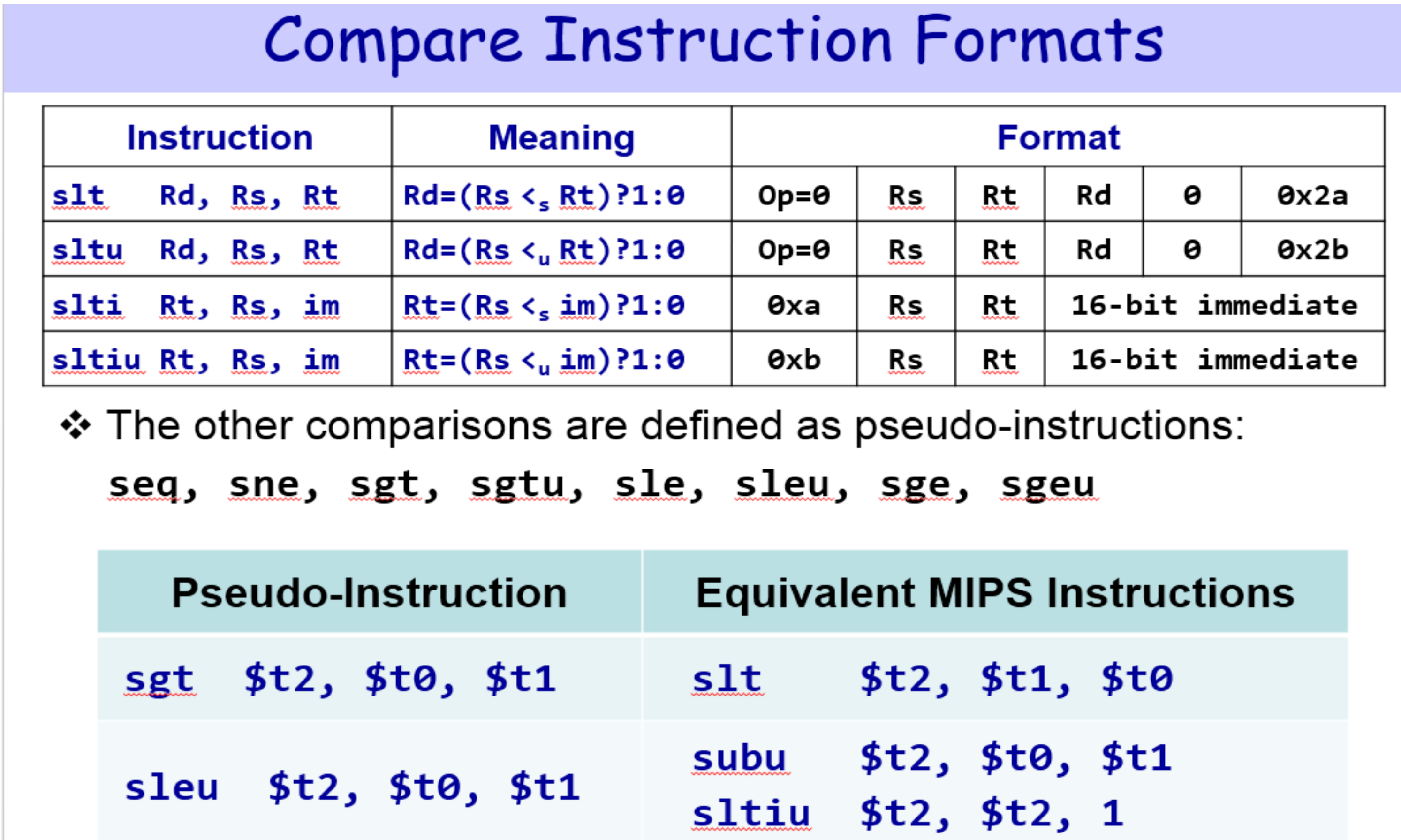


Signed vs. Unsigned

Signed comparison: slt, slti
Unsigned comparison: sltu, sltui

Example

```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

    slt  $t0, $s0, $s1  # signed
      (–1 < +1 ⇒ $t0 = 1)
    sltu $t0, $s0, $s1  # unsigned
      (+4,294,967,295 > +1 ⇒ $t0 = 0)
```



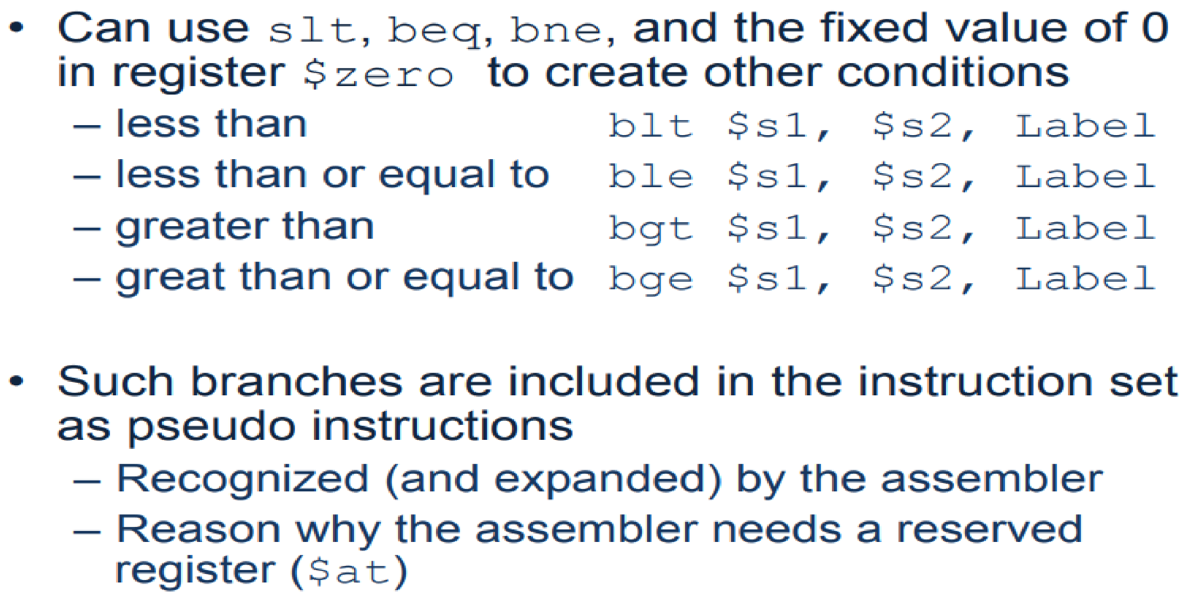
Compare Instruction Formats

Instruction	Meaning	Format					
`slt   Rd, Rs, Rt`	Rd=(Rs $<_s$ Rt)?1:0	Op=0	Rs	Rt	Rd	0	0x2a
`sltu  Rd, Rs, Rt`	Rd=(Rs $<_u$ Rt)?1:0	Op=0	Rs	Rt	Rd	0	0x2b
`slti  Rt, Rs, im`	Rt=(Rs $<_s$ im)?1:0	0xa	Rs	Rt	16-bit immediate		
`sltiu Rt, Rs, im`	Rt=(Rs $<_u$ im)?1:0	0xb	Rs	Rt	16-bit immediate		

❖ The other comparisons are defined as pseudo-instructions:
seq, sne, sgt, sgtu, sle, sleu, sge, sgeu

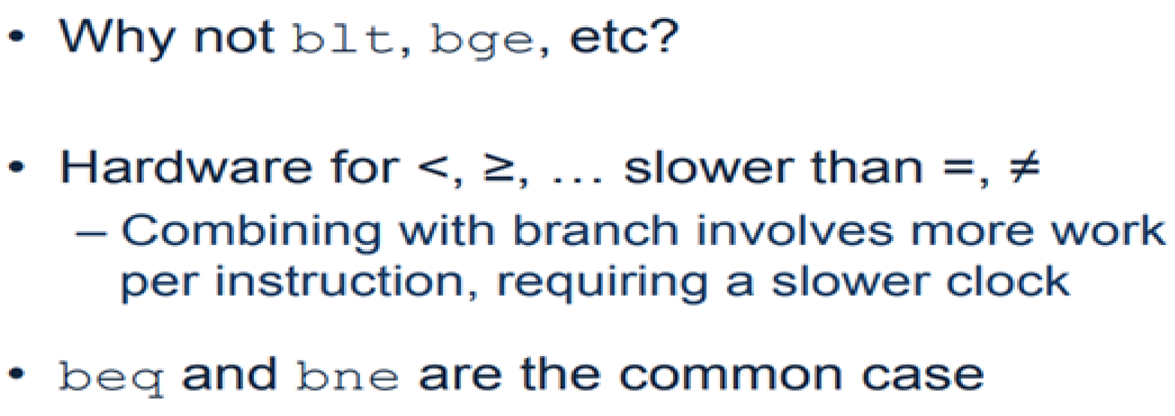
Pseudo-Instruction	Equivalent MIPS Instructions
`sgt  $t2, $t0, $t1`	`slt    $t2, $t1, $t0`

`sleu $t2, $t0, $t1`	`subu   $t2, $t0, $t1`
	`sltiu  $t2, $t2, 1`

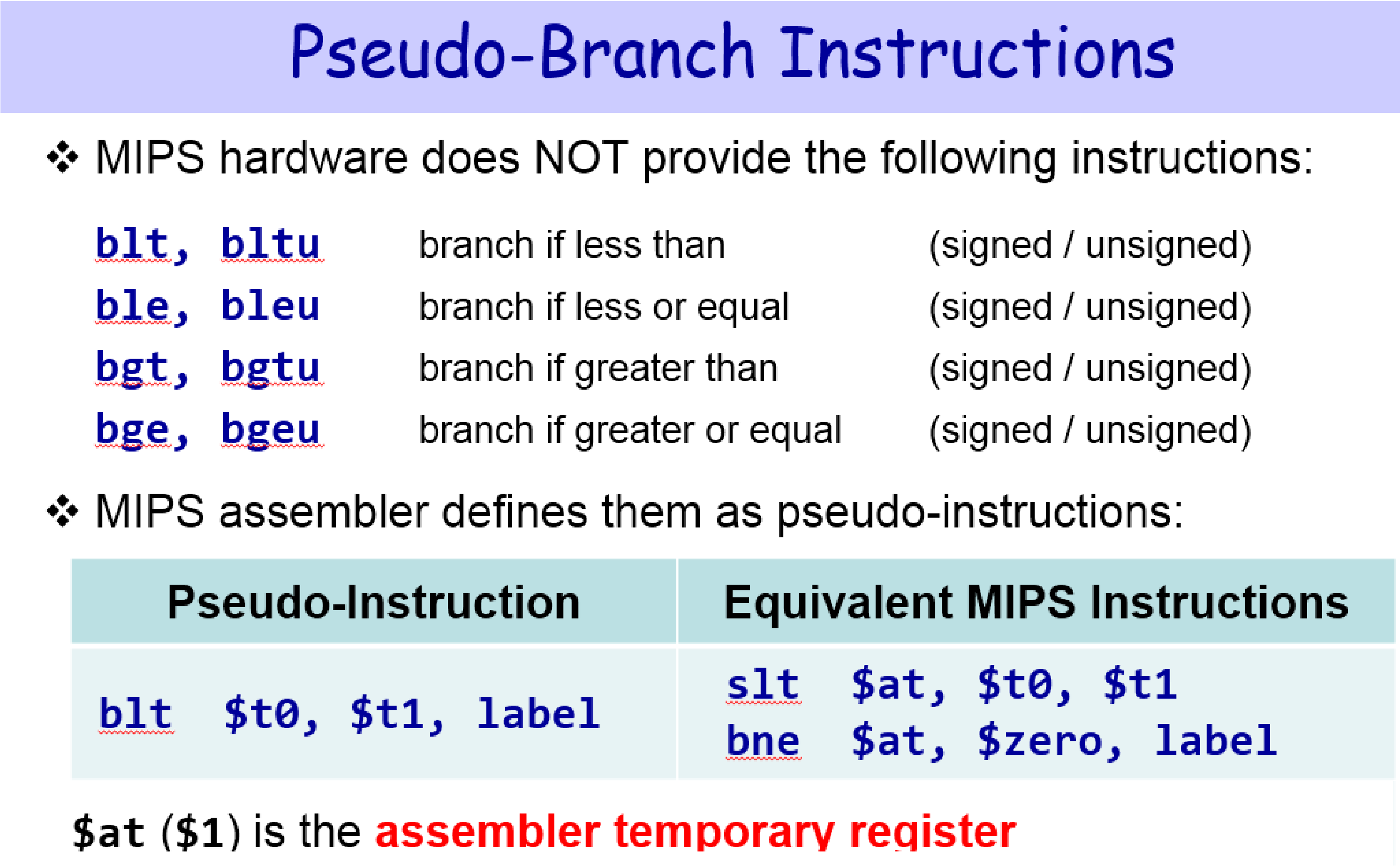


- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than `blt $s1, $s2, Label`
  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions
  - Recognized (and expanded) by the assembler
  - Reason why the assembler needs a reserved register (`$at`)



- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
- `beq` and `bne` are the common case



Pseudo-Branch Instructions

❖ MIPS hardware does NOT provide the following instructions:

`blt, bltu`	branch if less than	(signed / unsigned)
`ble, bleu`	branch if less or equal	(signed / unsigned)
`bgt, bgtu`	branch if greater than	(signed / unsigned)
`bge, bgeu`	branch if greater or equal	(signed / unsigned)

❖ MIPS assembler defines them as pseudo-instructions:

Pseudo-Instruction	Equivalent MIPS Instructions

`blt $t0, $t1, label`	`slt $at, $t0, $t1`
	`bne $at, $zero, label`

`$at` (`$1`) is the **assembler temporary register**

Example of one-to-one pseudoinstruction: The following

```
not   $s0              # complement ($s0)
```

is converted to the real instruction:

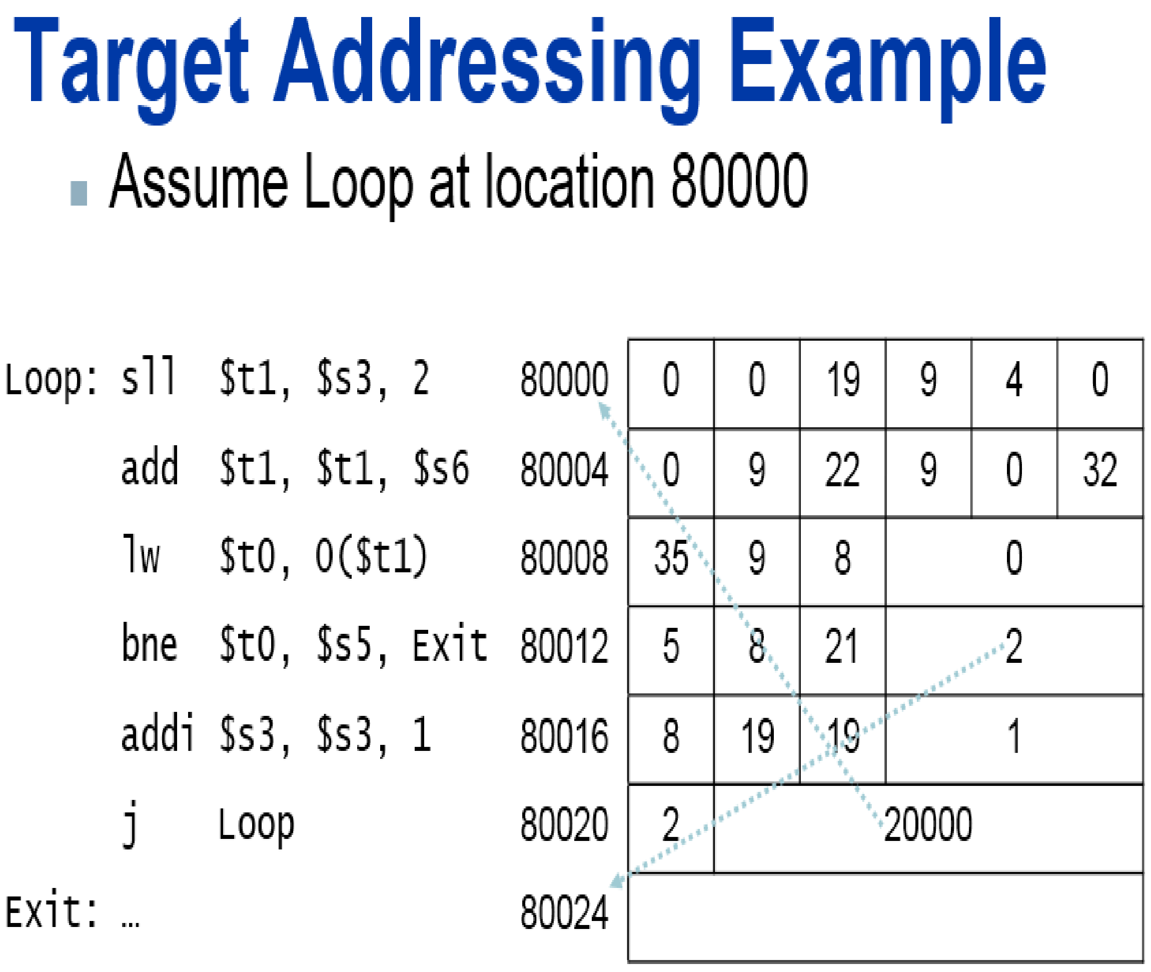
```
nor   $s0,$s0,$zero    # complement ($s0)
```

Example of one-to-several pseudoinstruction: The following

```
abs   $t0,$s0          # put |($s0)| into $t0
```

is converted to the sequence of real instructions:

```
add   $t0,$s0,$zero    # copy x into $t0
slt   $at,$t0,$zero    # is x negative?
beq   $at,$zero,+4     # if not, skip next instr
sub   $t0,$zero,$s0    # the result is 0 − x
```



Target Addressing Example

- Assume Loop at location 80000

Loop: sll $t1, $s3, 2	80000	0	0	19	9	4	0
add $t1, $t1, $s6	80004	0	9	22	9	0	32
lw $t0, 0($t1)	80008	35	9	8	0		
bne $t0, $s5, Exit	80012	5	8	21	2		
addi $s3, $s3, 1	80016	8	19	19	1		
j Loop	80020	2	20000				
Exit: …	80024						

Branch on equal

```
beq rs, rt, label
```

4	rs	rt	Offset
6	5	5	16

Conditionally branch the number of instructions specified by the offset if register `rs` equals `rt`.

Branch on not equal

```
bne rs, rt, label
```

5	rs	rt	Offset
6	5	5	16

Conditionally branch the number of instructions specified by the offset if register `rs` is not equal to `rt`.

Jump

```
j target
```

2	target
6	26

Unconditionally jump to the instruction at target.

Jump and link

```
jal target
```

3	target
6	26

Unconditionally jump to the instruction at target. Save the address of the next instruction in register `$ra`.

Jump and link register

```
jalr rs, rd
```

0	rs	0	rd	0	9
6	5	5	5	5	6

Unconditionally jump to the instruction whose address is in register `rs`. Save the address of the next instruction in register `rd` (which defaults to 31).

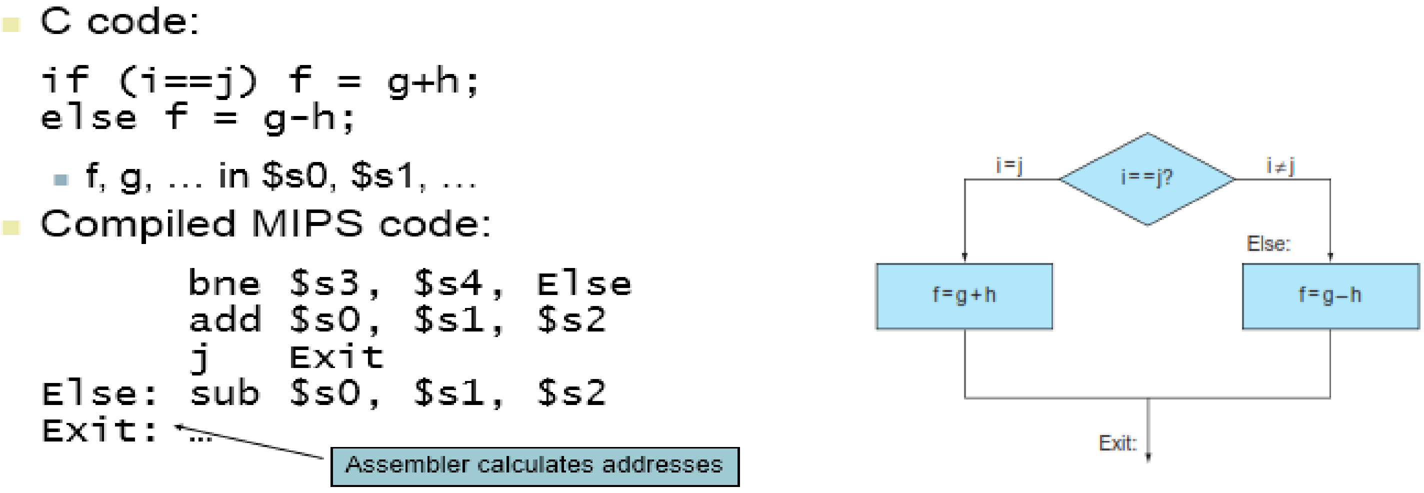
Jump register

```
jr rs
```

0	rs	0	8
6	5	15	6

Unconditionally jump to the instruction whose address is in register `rs`.

Compiling IF statement



- C code:

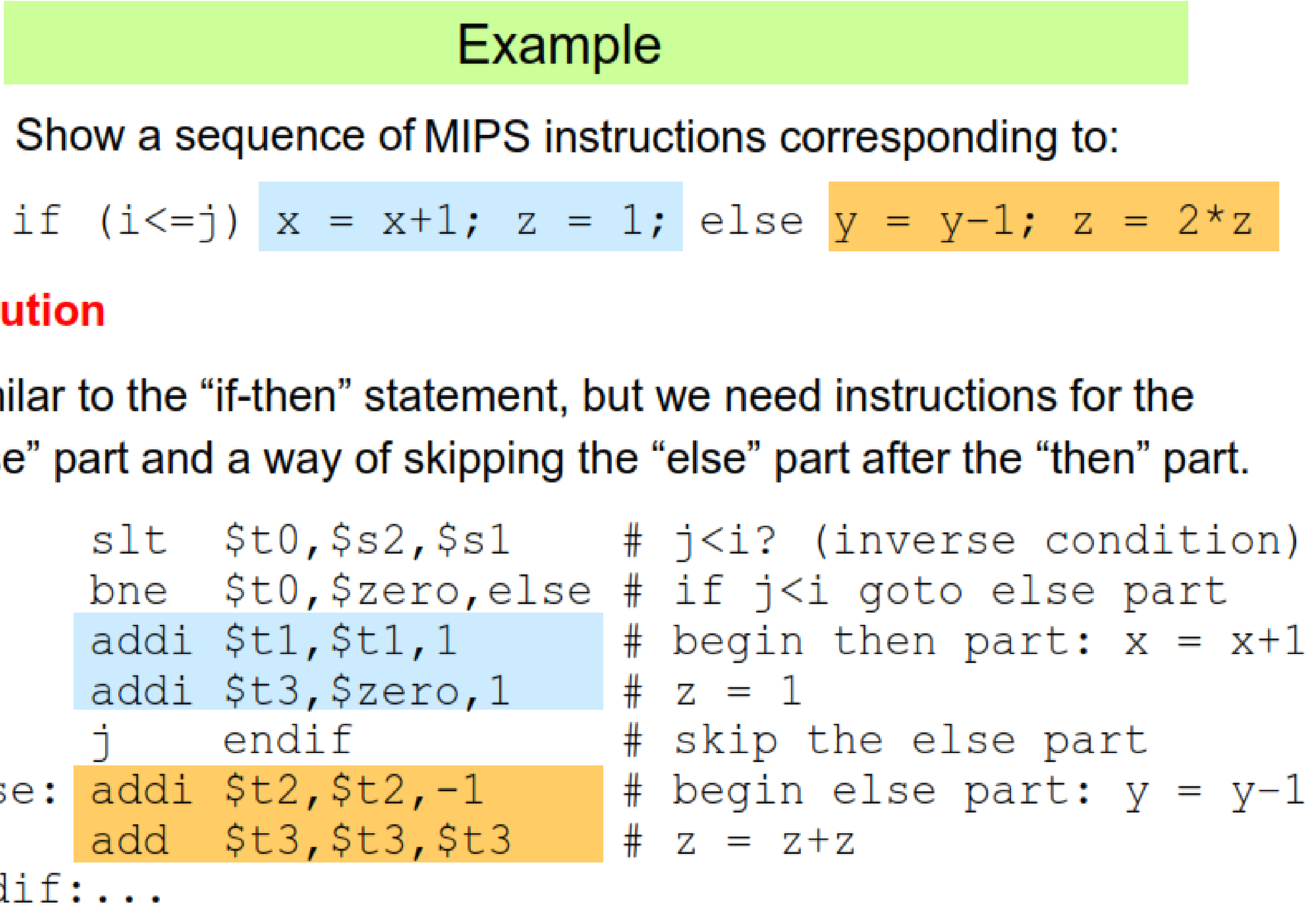
```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in $s0, $s1, …

- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses



Example

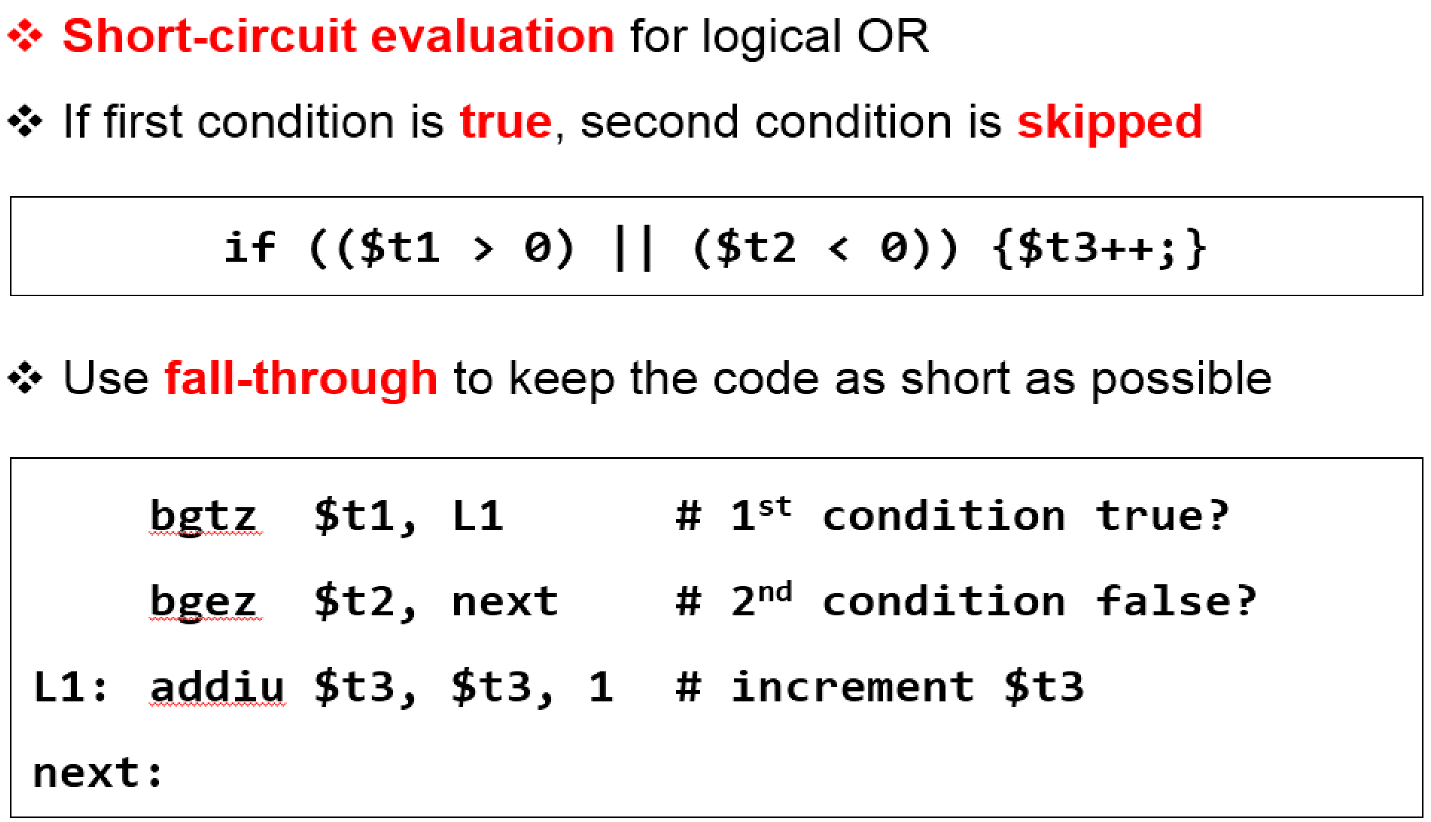
Show a sequence of MIPS instructions corresponding to:

```
if (i<=j) x = x+1; z = 1; else y = y-1; z = 2*z
```

Solution

Similar to the "if-then" statement, but we need instructions for the "else" part and a way of skipping the "else" part after the "then" part.

```
      slt  $t0,$s2,$s1     # j<i? (inverse condition)
      bne  $t0,$zero,else  # if j<i goto else part
      addi $t1,$t1,1       # begin then part: x = x+1
      addi $t3,$zero,1     # z = 1
      j    endif           # skip the else part
else: addi $t2,$t2,-1      # begin else part: y = y-1
      add  $t3,$t3,$t3     # z = z+z
endif:...
```

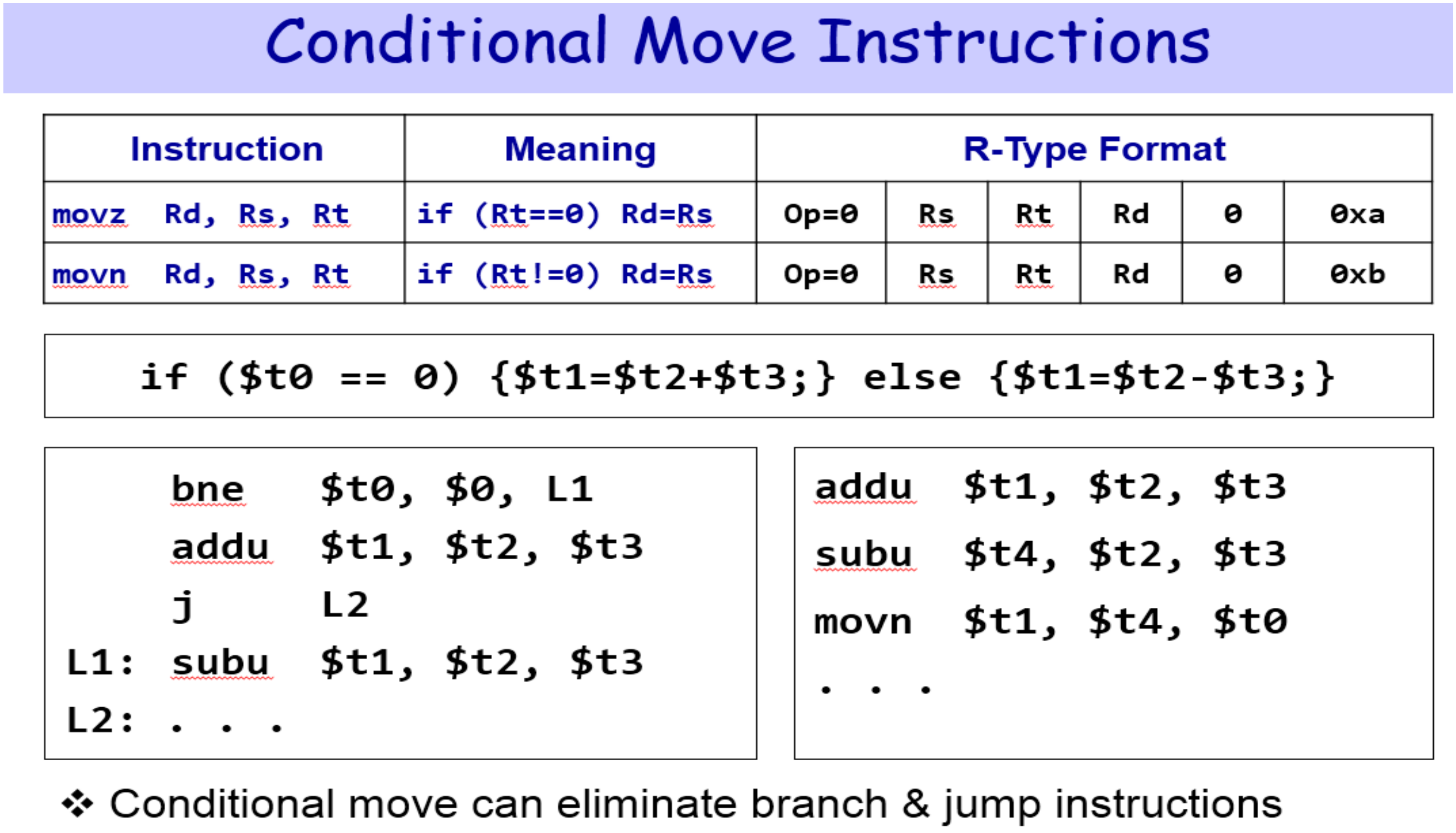


- **Short-circuit evaluation** for logical OR
- If first condition is **true**, second condition is **skipped**

```
if (($t1 > 0) || ($t2 < 0)) {$t3++;}
```

- Use **fall-through** to keep the code as short as possible

```
    bgtz  $t1, L1          # 1st condition true?
    bgez  $t2, next        # 2nd condition false?
L1: addiu $t3, $t3, 1      # increment $t3
next:
```



Conditional Move Instructions

Instruction	Meaning	R-Type Format					
`movz Rd, Rs, Rt`	`if (Rt==0) Rd=Rs`	Op=0	Rs	Rt	Rd	0	0xa
`movn Rd, Rs, Rt`	`if (Rt!=0) Rd=Rs`	Op=0	Rs	Rt	Rd	0	0xb

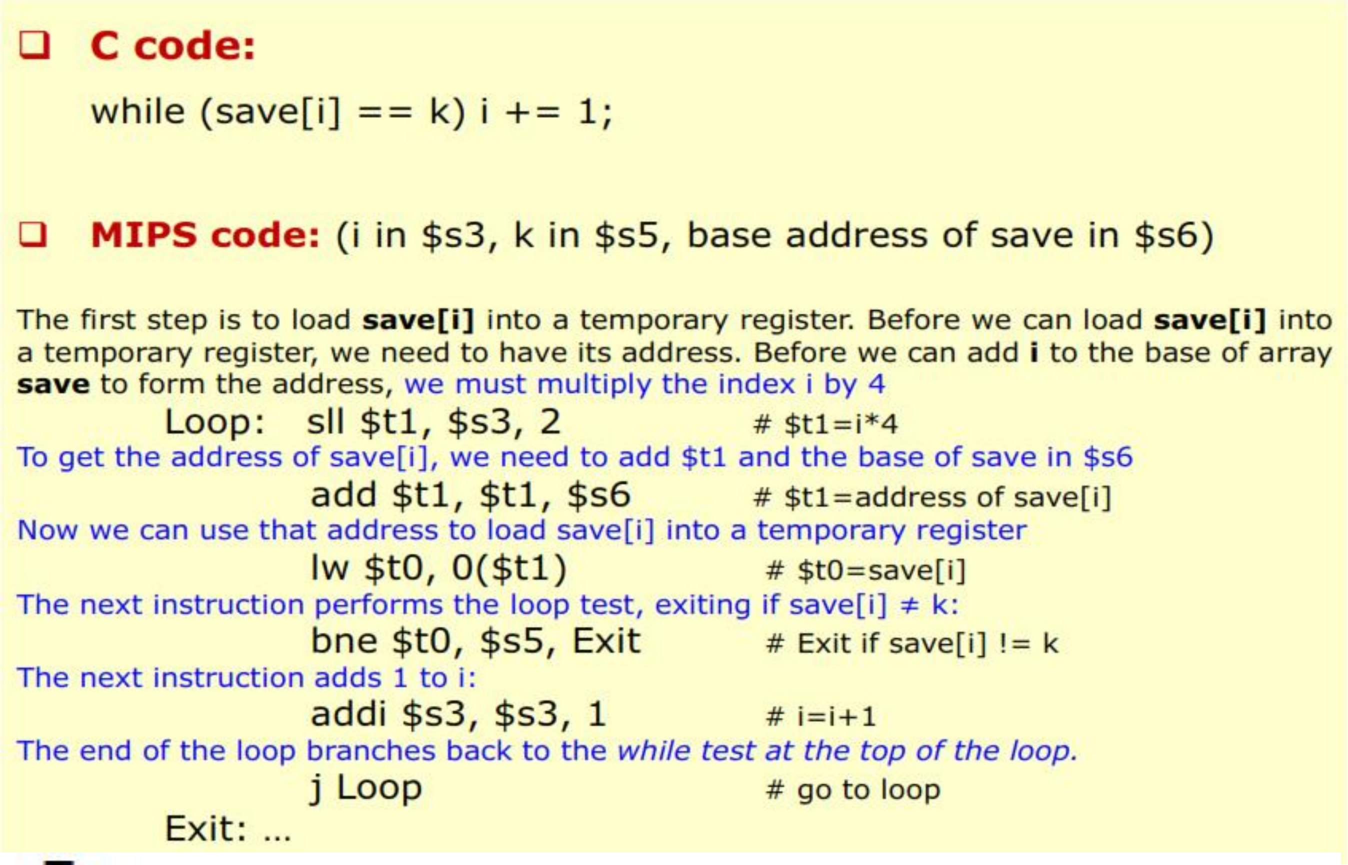
```
if ($t0 == 0) {$t1=$t2+$t3;} else {$t1=$t2-$t3;}
```

```
    bne   $t0, $0, L1
    addu  $t1, $t2, $t3
    j     L2
L1: subu  $t1, $t2, $t3
L2: . . .
```

```
addu  $t1, $t2, $t3
subu  $t4, $t2, $t3
movn  $t1, $t4, $t0
. . .
```

- Conditional move can eliminate branch & jump instructions

Compiling LOOP statement



❑ **C code:**

```
while (save[i] == k) i += 1;
```

❑ **MIPS code:** (i in $s3, k in $s5, base address of save in $s6)

The first step is to load **save[i]** into a temporary register. Before we can load **save[i]** into a temporary register, we need to have its address. Before we can add **i** to the base of array **save** to form the address, we must multiply the index i by 4

```
        Loop:   sll $t1, $s3, 2              # $t1=i*4
```

To get the address of save[i], we need to add $t1 and the base of save in $s6

```
                add $t1, $t1, $s6            # $t1=address of save[i]
```

Now we can use that address to load save[i] into a temporary register

```
                lw $t0, 0($t1)               # $t0=save[i]
```

The next instruction performs the loop test, exiting if save[i] ≠ k:

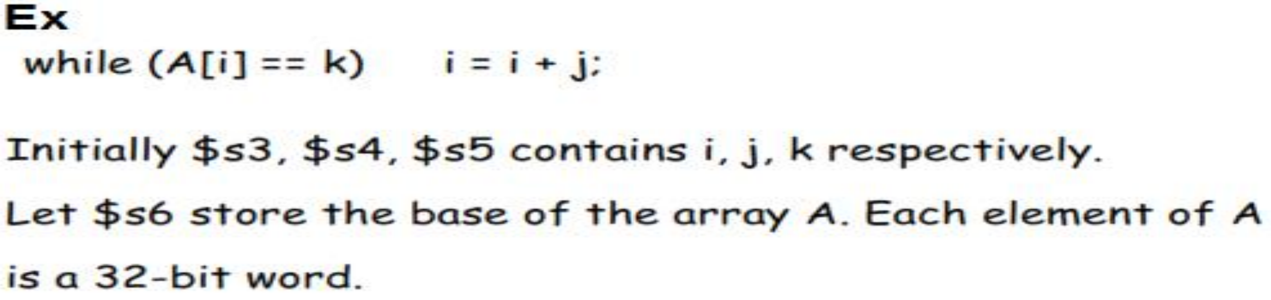
```
                bne $t0, $s5, Exit           # Exit if save[i] != k
```

The next instruction adds 1 to i:

```
                addi $s3, $s3, 1             # i=i+1
```

The end of the loop branches back to the *while test at the top of the loop.*

```
                j Loop                       # go to loop
        Exit: ...
```

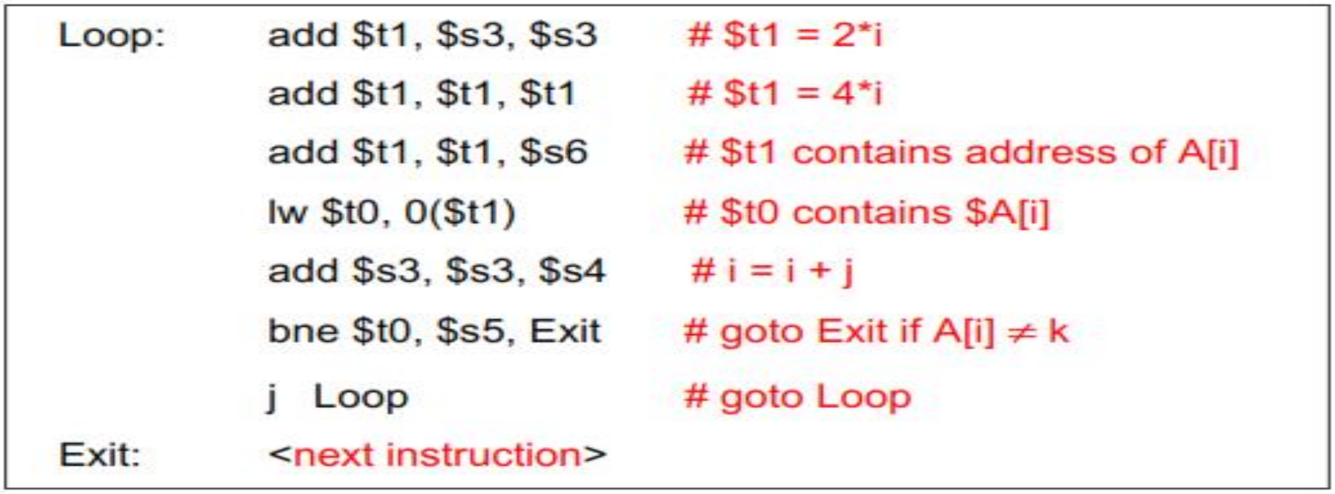


Ex

```
while (A[i] == k)      i = i + j;
```

Initially $s3, $s4, $s5 contains i, j, k respectively.

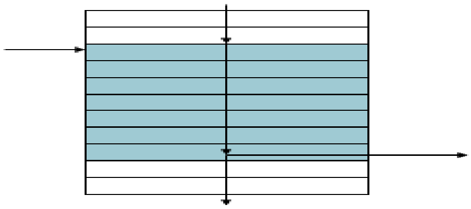
Let $s6 store the base of the array A. Each element of A is a 32-bit word.



```
Loop:     add $t1, $s3, $s3      # $t1 = 2*i
          add $t1, $t1, $t1      # $t1 = 4*i
          add $t1, $t1, $s6      # $t1 contains address of A[i]
          lw $t0, 0($t1)         # $t0 contains $A[i]
          add $s3, $s3, $s4      # i = i + j
          bne $t0, $s5, Exit     # goto Exit if A[i] ≠ k
          j   Loop               # goto Loop
Exit:     <next instruction>
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Supporting Procedures in Computer Hardware

❖ **A function (or a procedure)** is a block of instructions that can be called at several different points in the program

✧ **Allows the programmer to focus on just one task at a time**

✧ **Allows code to be reused**

✧ **Reduce duplication of code and enable reuse.**

❖ **The function that initiates the call is known as the caller**

❖ **The function that receives the call is known as the callee**

❖ **When the callee finishes execution, control is transferred back to the caller function.**

❖ **A function can receive parameters and return results**

❖ **The function parameters and results act as an interface between a function and the rest of the program**

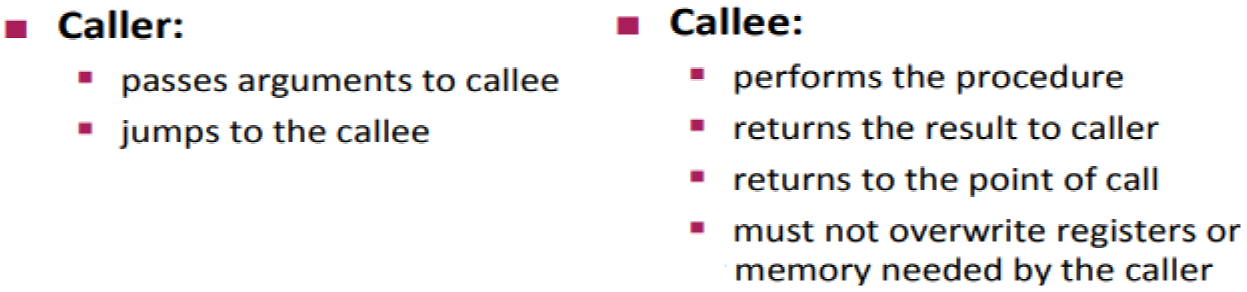
❖ **To execution a function, the caller does the following:**

- Puts the parameters in a place that can be accessed by the callee
- Transfer control to the callee function

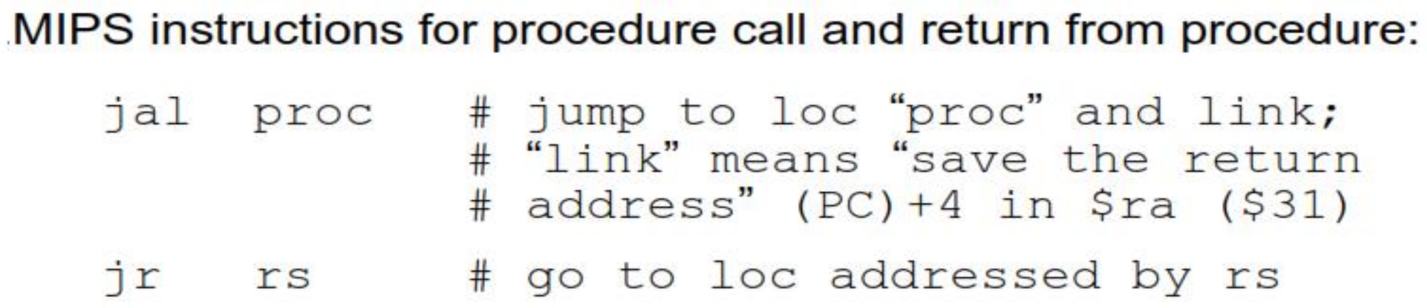
❖ **To return from a function, the callee does the following:**

- Puts the results in a place that can be accessed by the caller

- Return control to the caller, next to where the function call was made

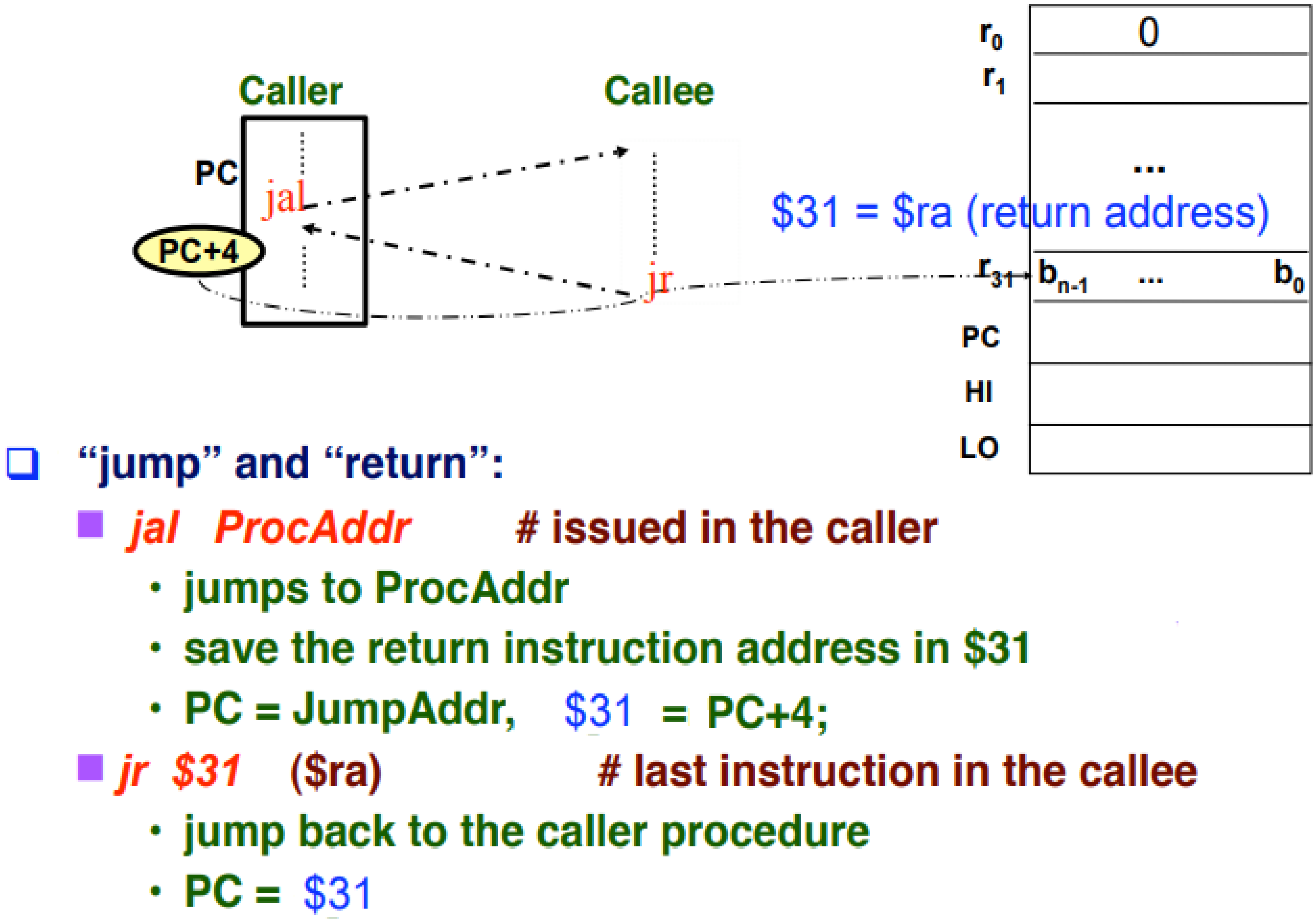


- **Caller:**
  - passes arguments to callee
  - jumps to the callee
- **Callee:**
  - performs the procedure
  - returns the result to caller
  - returns to the point of call
  - must not overwrite registers or memory needed by the caller



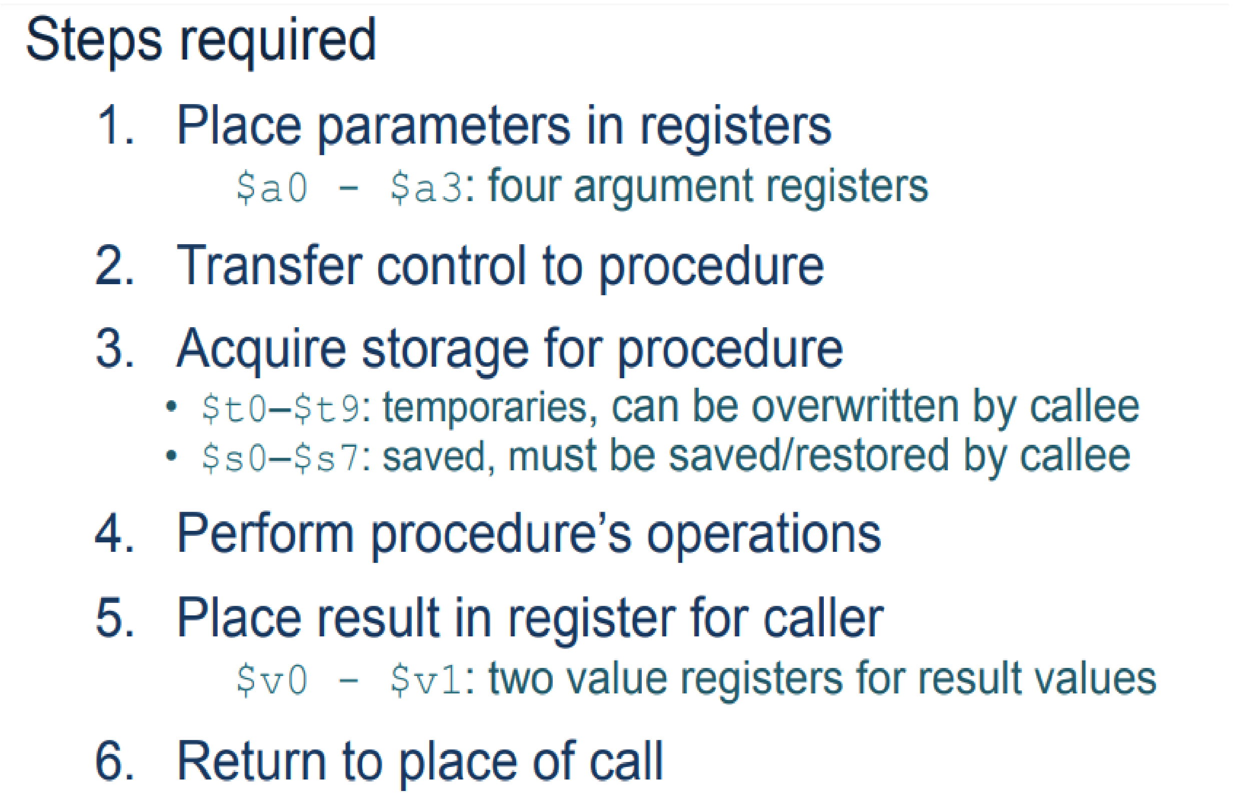
MIPS instructions for procedure call and return from procedure:

```
jal   proc    # jump to loc "proc" and link;
              # "link" means "save the return
              # address" (PC)+4 in $ra ($31)

jr    rs      # go to loc addressed by rs
```



❑ **"jump" and "return":**

- ***jal ProcAddr*** **# issued in the caller**
  - **jumps to ProcAddr**
  - **save the return instruction address in $31**
  - **PC = JumpAddr, $31 = PC+4;**
- ***jr $31*** **($ra)** **# last instruction in the callee**
  - **jump back to the caller procedure**
  - **PC = $31**



Steps required

1. Place parameters in registers

   `$a0 – $a3`: four argument registers

2. Transfer control to procedure

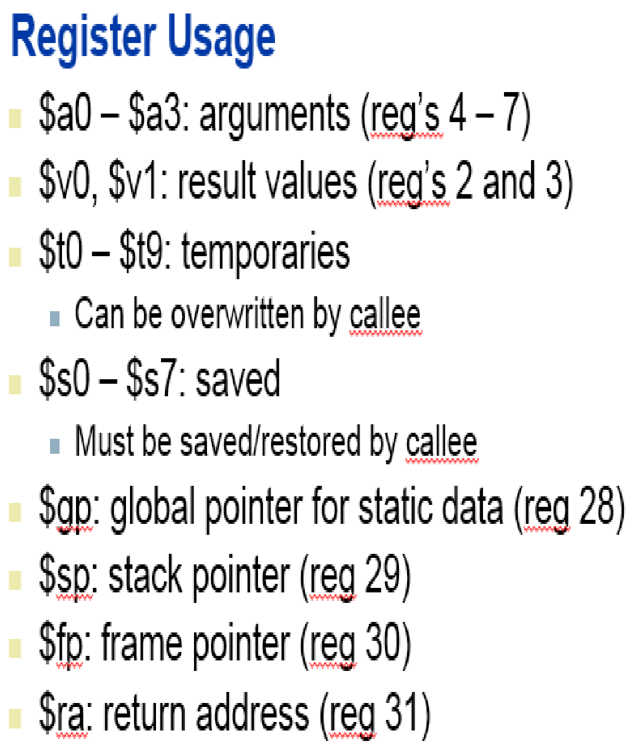
3. Acquire storage for procedure
   - `$t0–$t9`: temporaries, can be overwritten by callee
   - `$s0–$s7`: saved, must be saved/restored by callee

4. Perform procedure's operations

5. Place result in register for caller

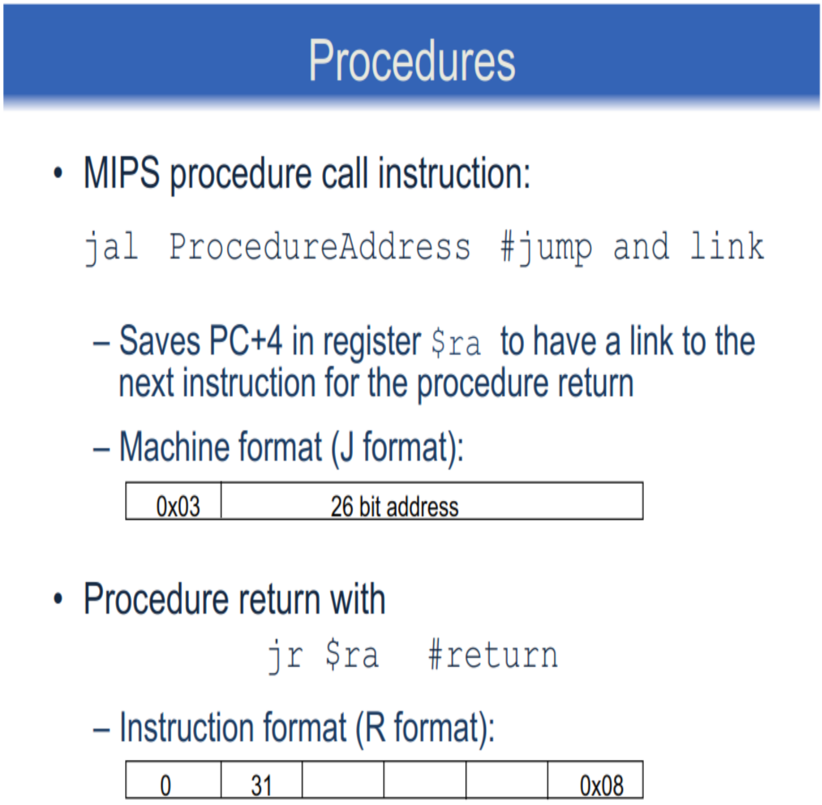
   `$v0 – $v1`: two value registers for result values

6. Return to place of call



Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)



Procedures

- MIPS procedure call instruction:

```
jal  ProcedureAddress  #jump and link
```

  – Saves PC+4 in register `$ra` to have a link to the next instruction for the procedure return

  – Machine format (J format):

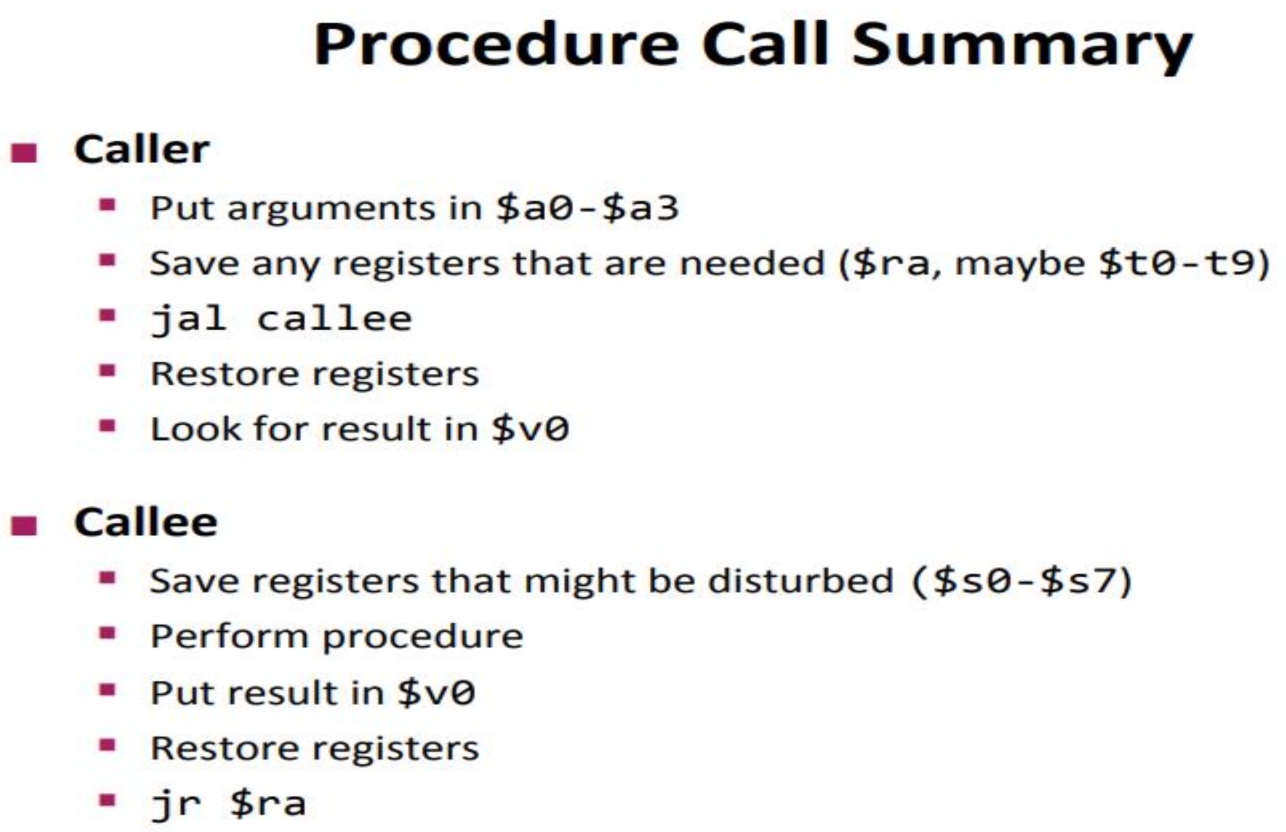
0x03	26 bit address

- Procedure return with

```
          jr $ra   #return
```

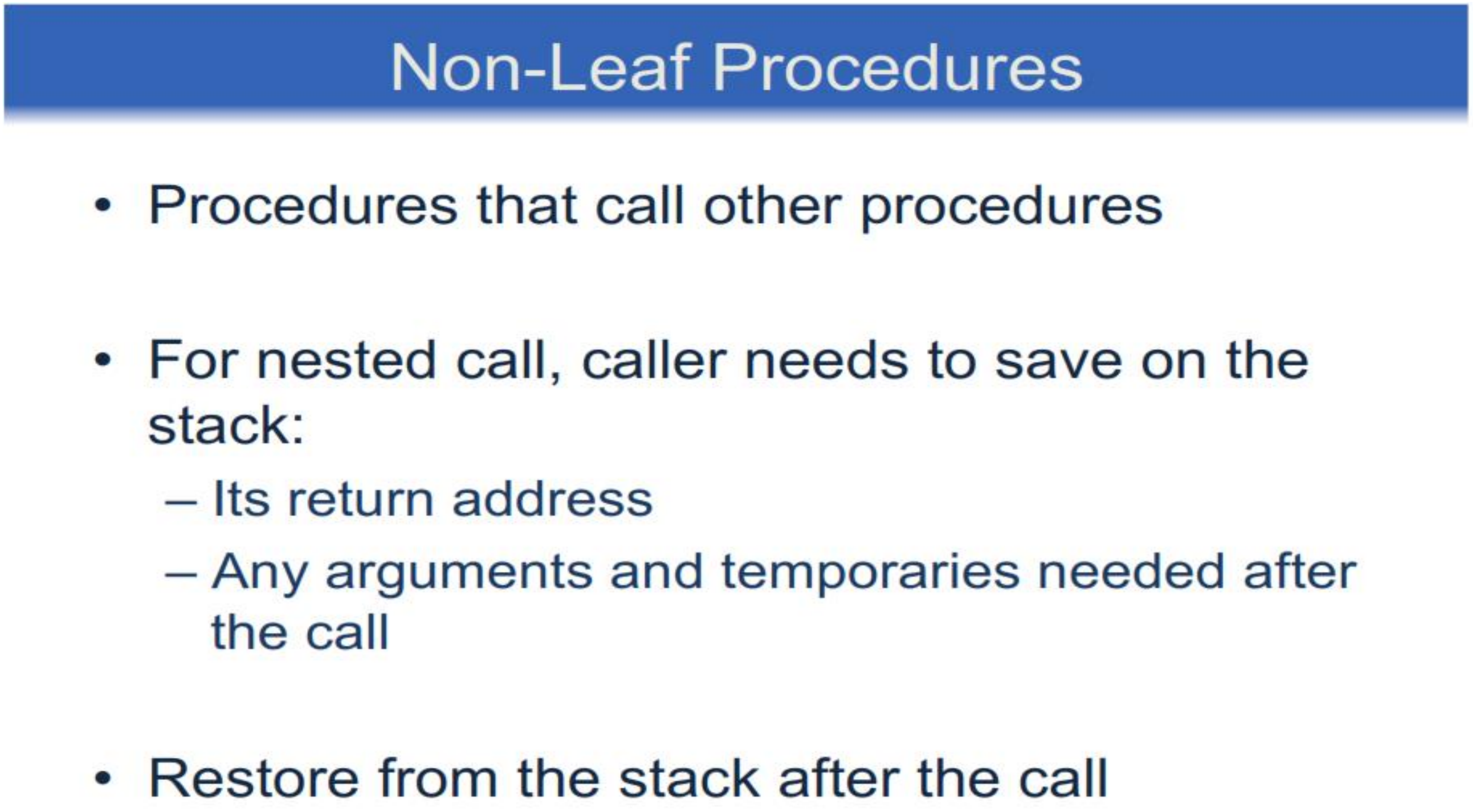
  – Instruction format (R format):

0	31				0x08



Procedure Call Summary

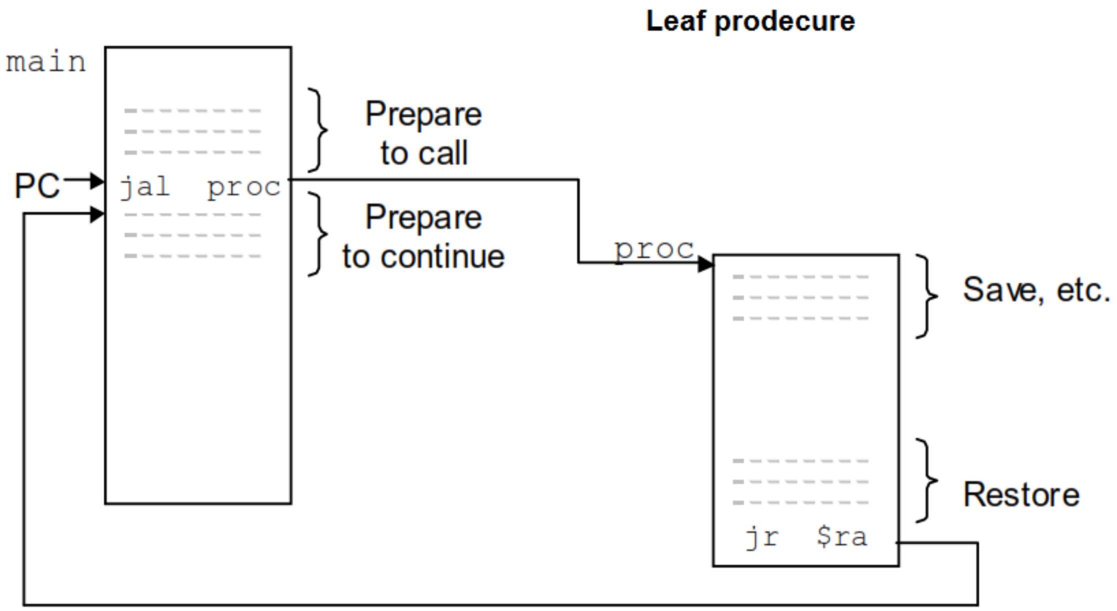
- **Caller**
  - Put arguments in `$a0-$a3`
  - Save any registers that are needed (`$ra`, maybe `$t0-t9`)
  - `jal callee`
  - Restore registers
  - Look for result in `$v0`
- **Callee**
  - Save registers that might be disturbed (`$s0-$s7`)
  - Perform procedure
  - Put result in `$v0`
  - Restore registers
  - `jr $ra`



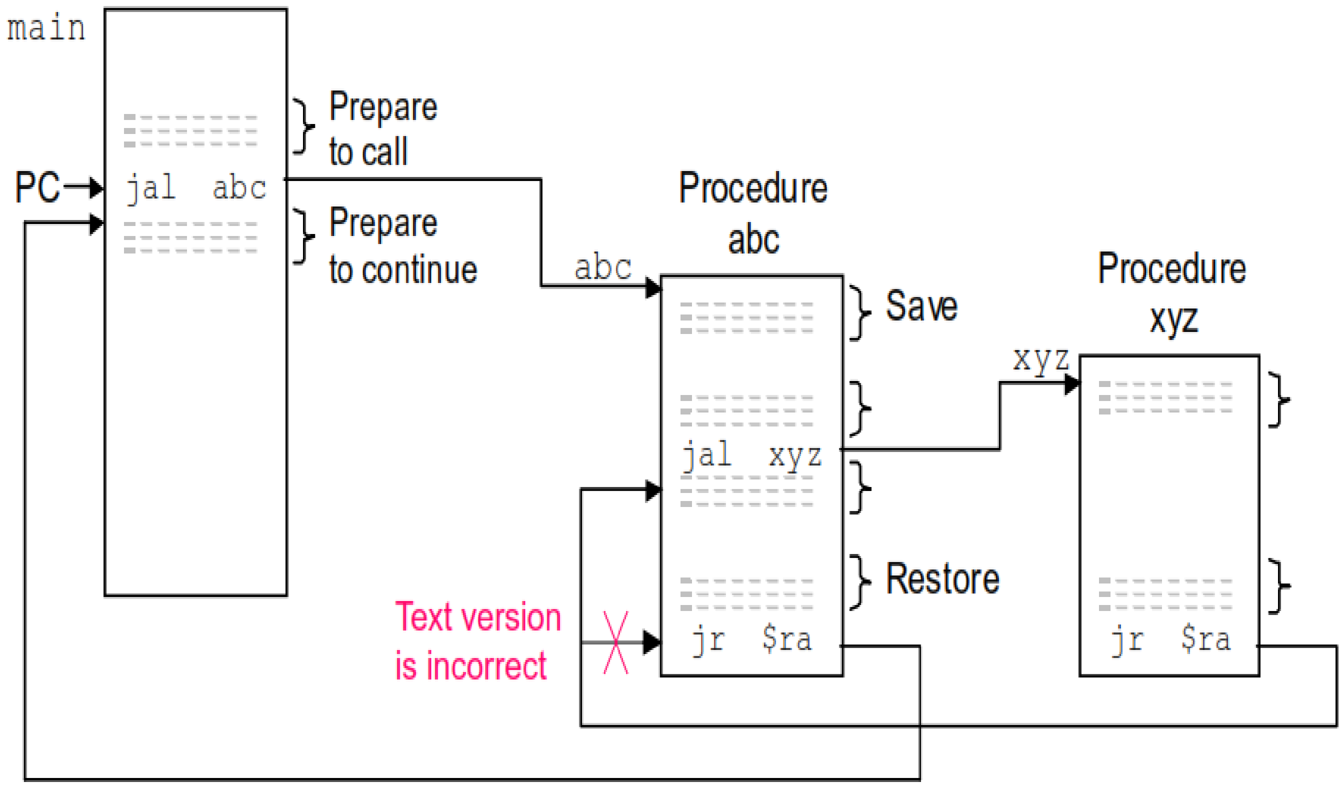
Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

main program and a procedure



Nested Procedure Calls

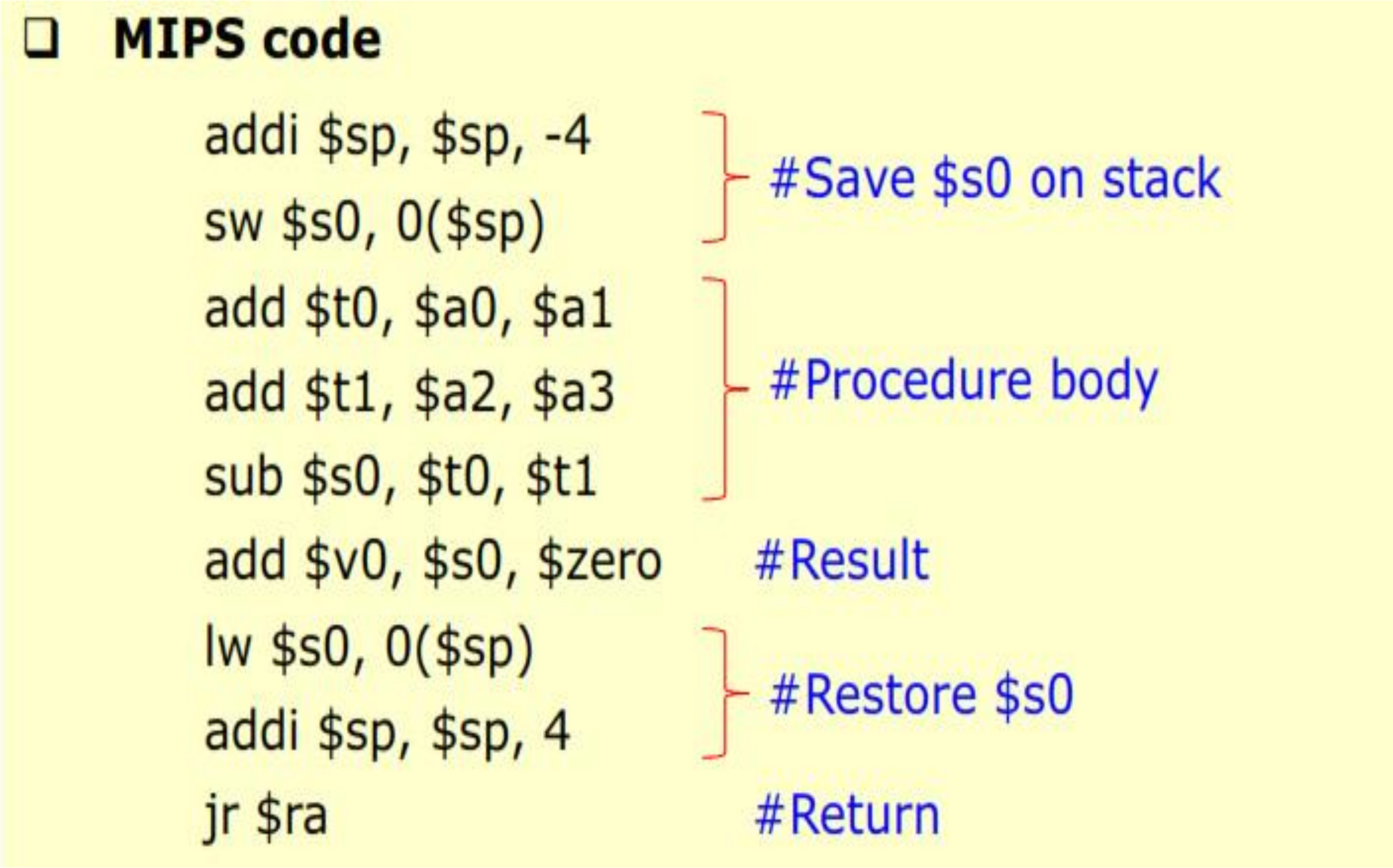


Leaf Procedure Example

- C code:

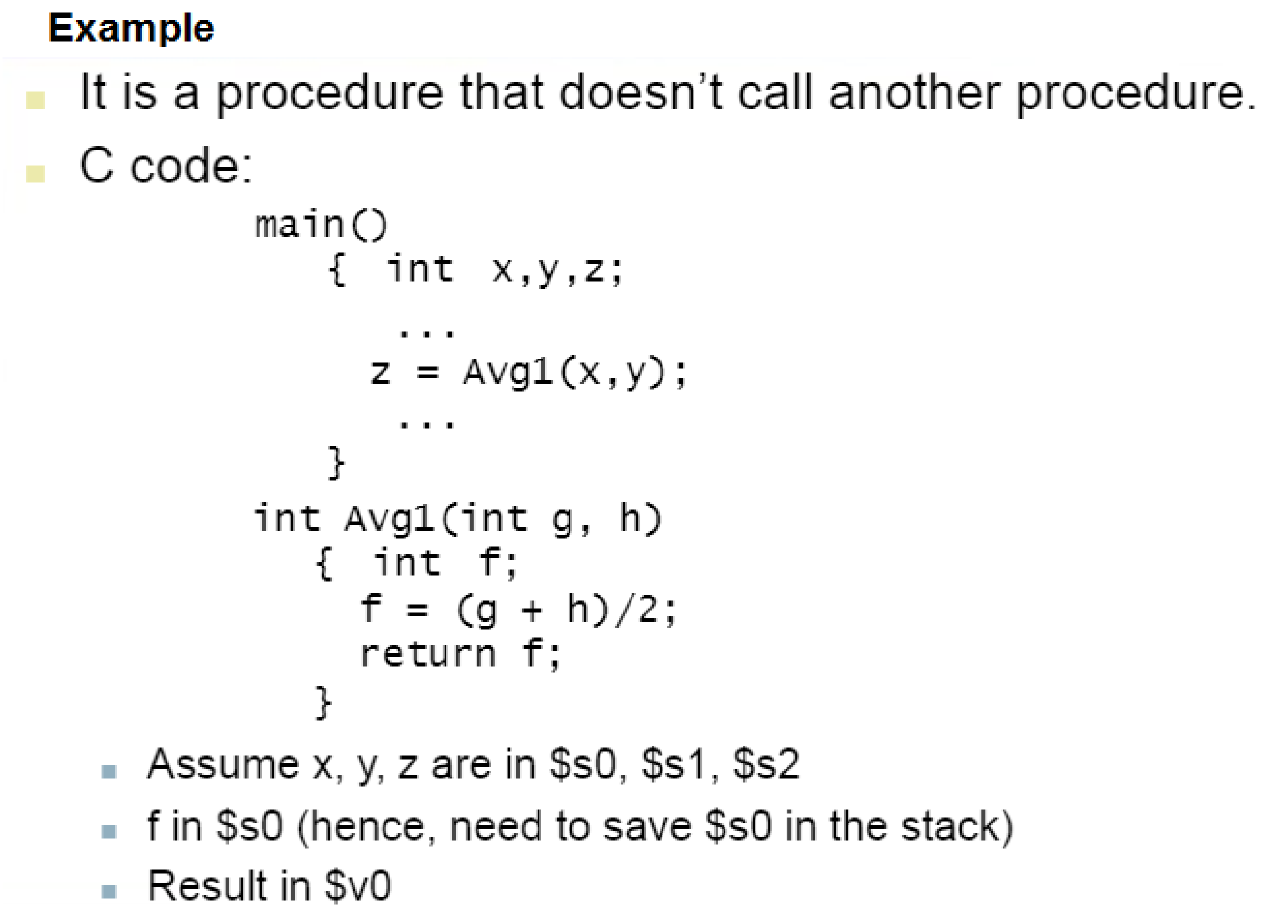
```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0



❑ **MIPS code**

```
addi $sp, $sp, -4
sw $s0, 0($sp)           #Save $s0 on stack
add $t0, $a0, $a1
add $t1, $a2, $a3        #Procedure body
sub $s0, $t0, $t1
add $v0, $s0, $zero      #Result
lw $s0, 0($sp)
addi $sp, $sp, 4         #Restore $s0
jr $ra                   #Return
```

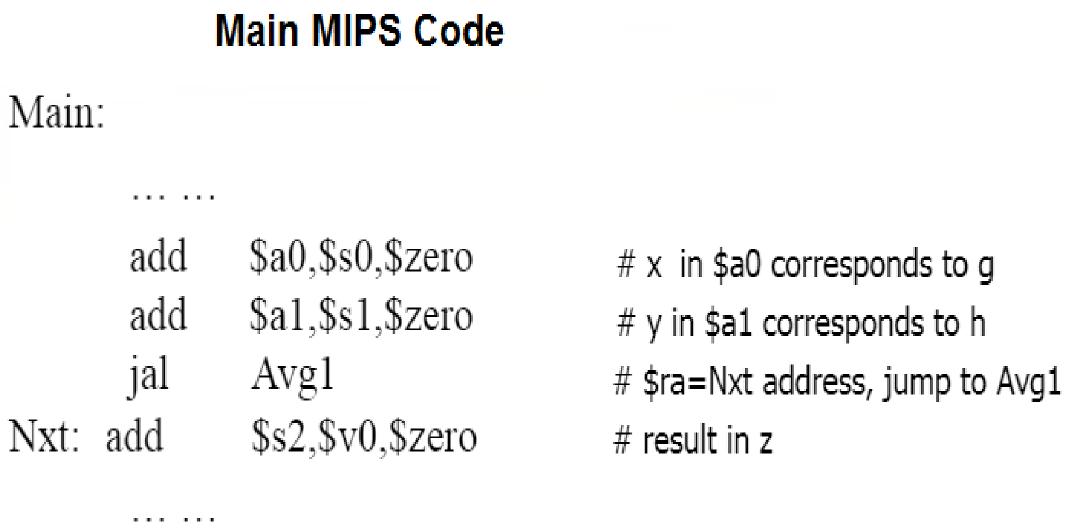


Example

- It is a procedure that doesn't call another procedure.
- C code:

```
main()
   {  int  x,y,z;
        ...
      z = Avg1(x,y);
        ...
   }
int Avg1(int g, h)
   {  int  f;
      f = (g + h)/2;
      return f;
   }
```

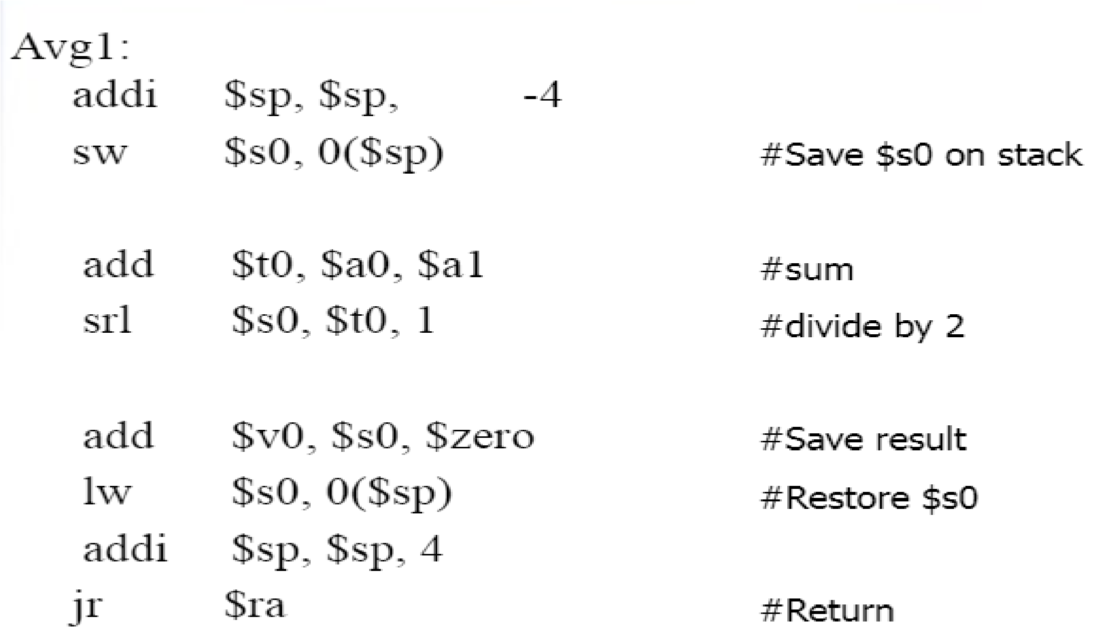
- Assume x, y, z are in $s0, $s1, $s2
- f in $s0 (hence, need to save $s0 in the stack)
- Result in $v0



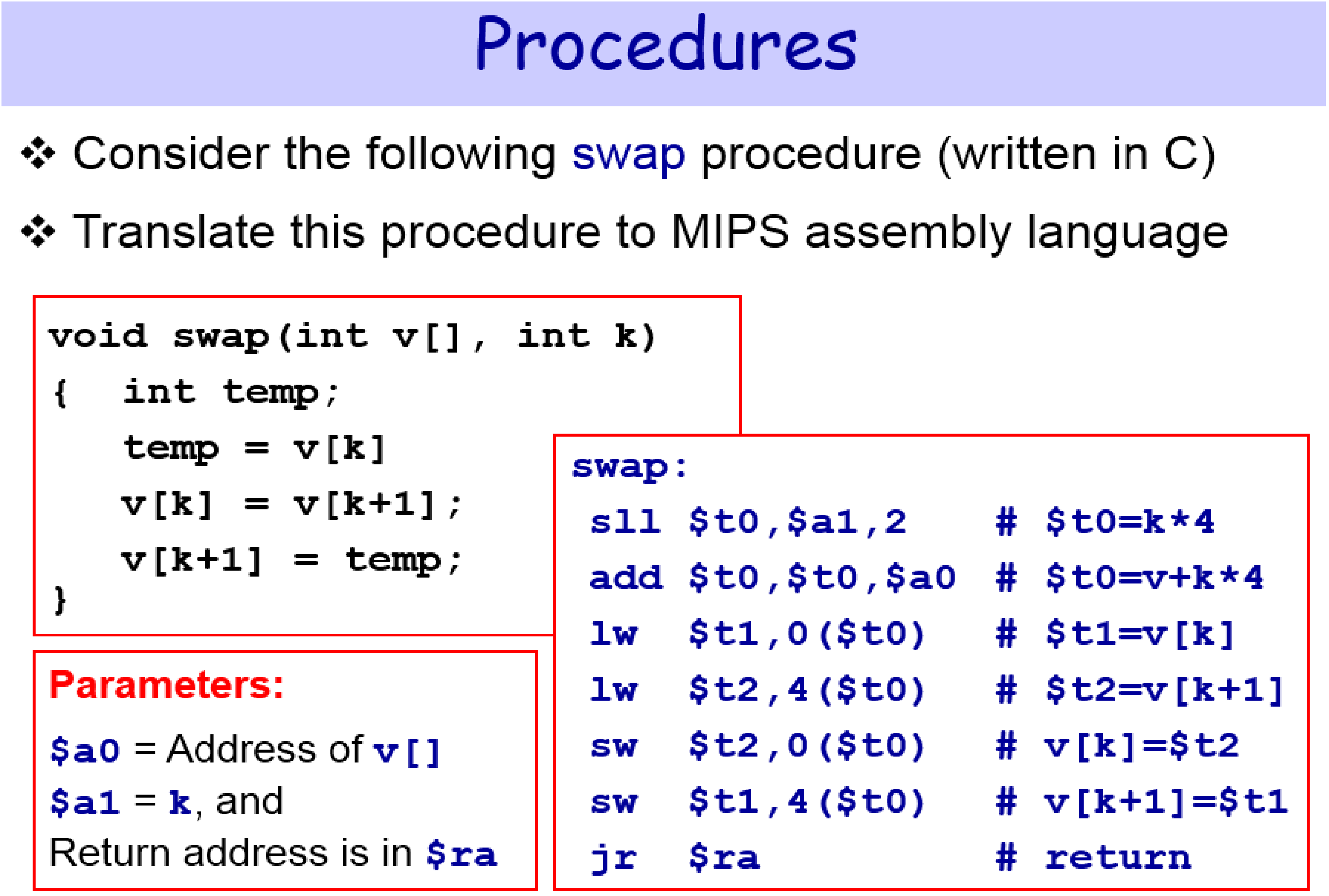
Main MIPS Code

```
Main:
          … …
          add    $a0,$s0,$zero        # x  in $a0 corresponds to g
          add    $a1,$s1,$zero        # y in $a1 corresponds to h
          jal    Avg1                 # $ra=Nxt address, jump to Avg1
Nxt:   add    $s2,$v0,$zero        # result in z
          … …
```

Leaf Procedure MIPS Code



```
Avg1:
   addi    $sp, $sp,       -4
   sw      $s0, 0($sp)                  #Save $s0 on stack

    add    $t0, $a0, $a1                #sum
    srl    $s0, $t0, 1                  #divide by 2

    add    $v0, $s0, $zero              #Save result
    lw     $s0, 0($sp)                  #Restore $s0
    addi   $sp, $sp, 4
   jr      $ra                          #Return
```



Procedures

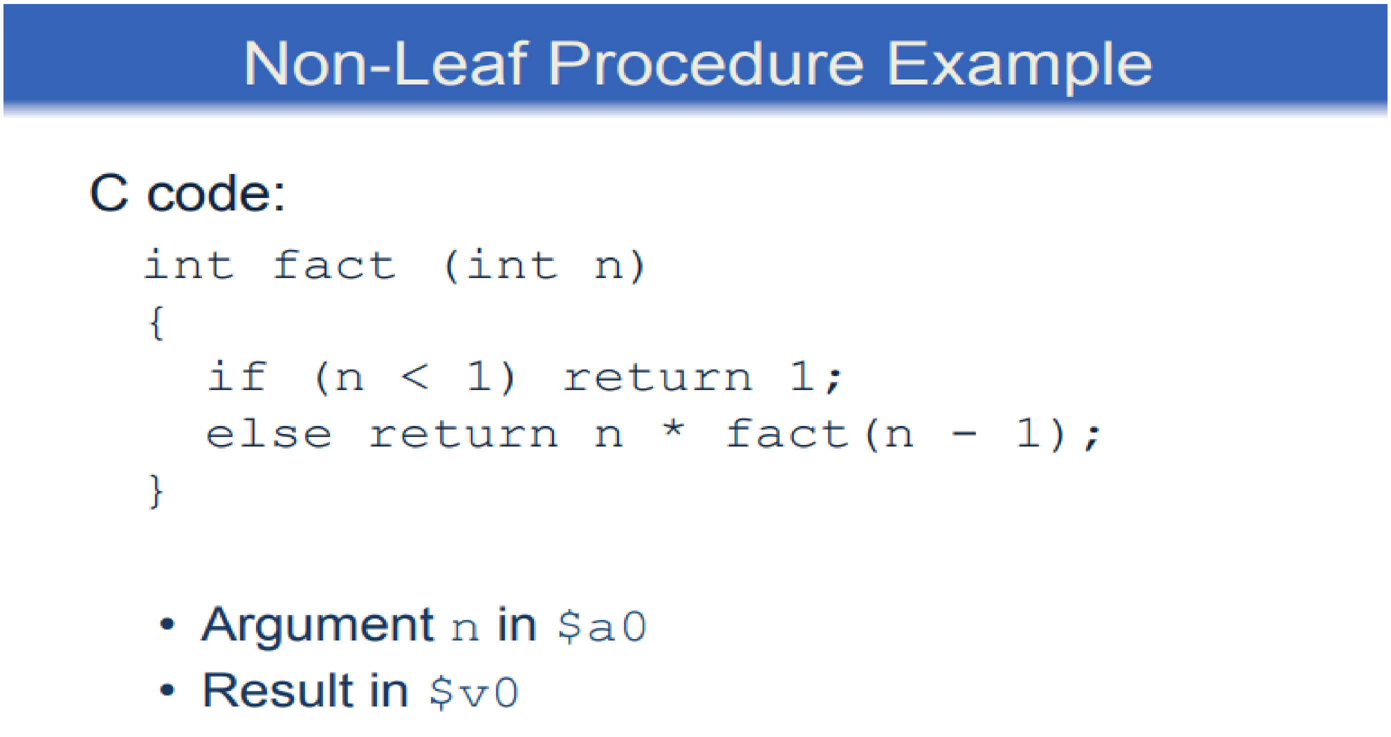
- Consider the following swap procedure (written in C)
- Translate this procedure to MIPS assembly language

```
void swap(int v[], int k)
{   int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Parameters:

$a0 = Address of v[]
$a1 = k, and
Return address is in $ra

```
swap:
 sll $t0,$a1,2      # $t0=k*4
 add $t0,$t0,$a0    # $t0=v+k*4
 lw  $t1,0($t0)     # $t1=v[k]
 lw  $t2,4($t0)     # $t2=v[k+1]
 sw  $t2,0($t0)     # v[k]=$t2
 sw  $t1,4($t0)     # v[k+1]=$t1
 jr  $ra            # return
```

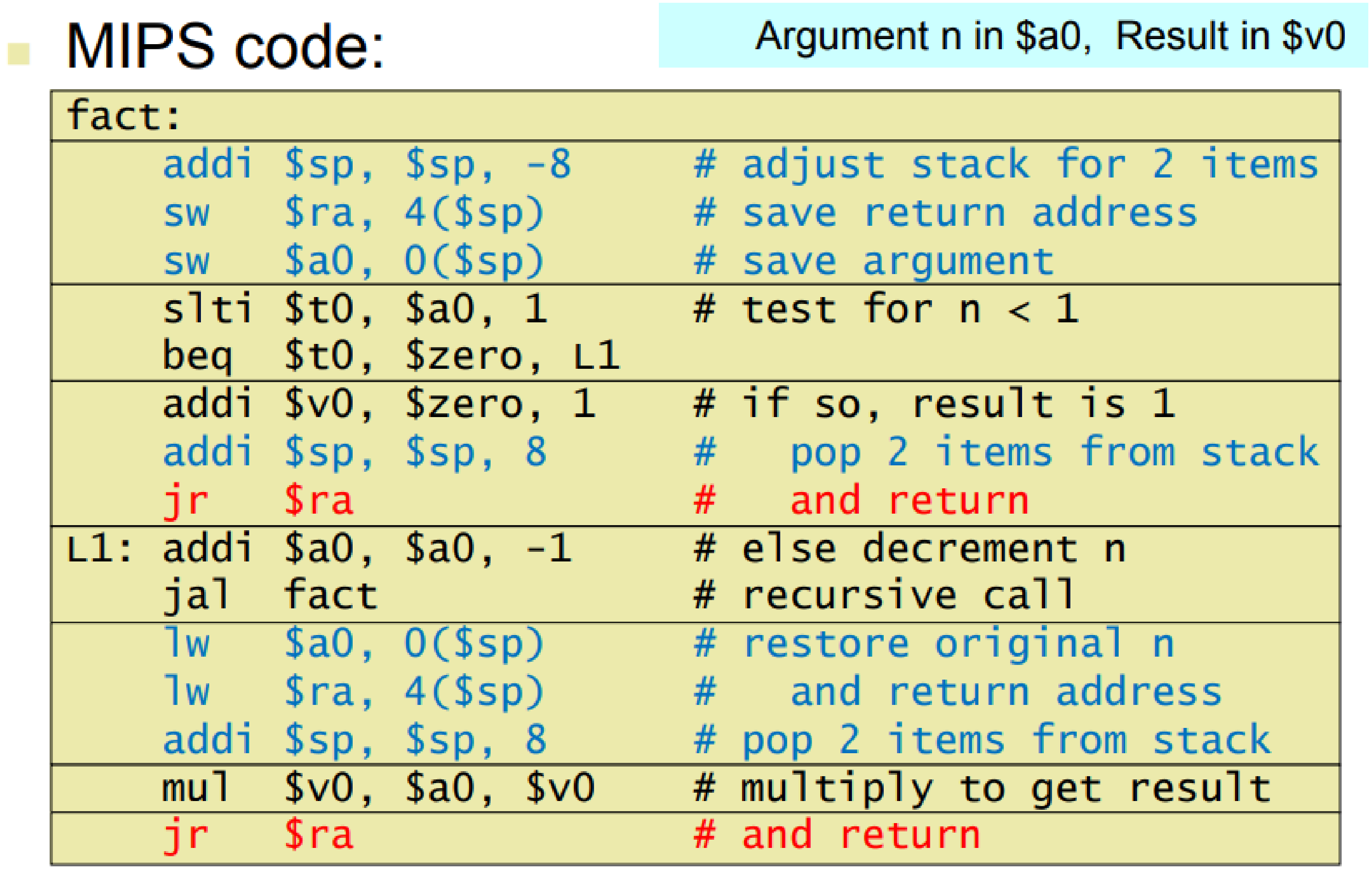


Non-Leaf Procedure Example

C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

- Argument n in $a0
- Result in $v0

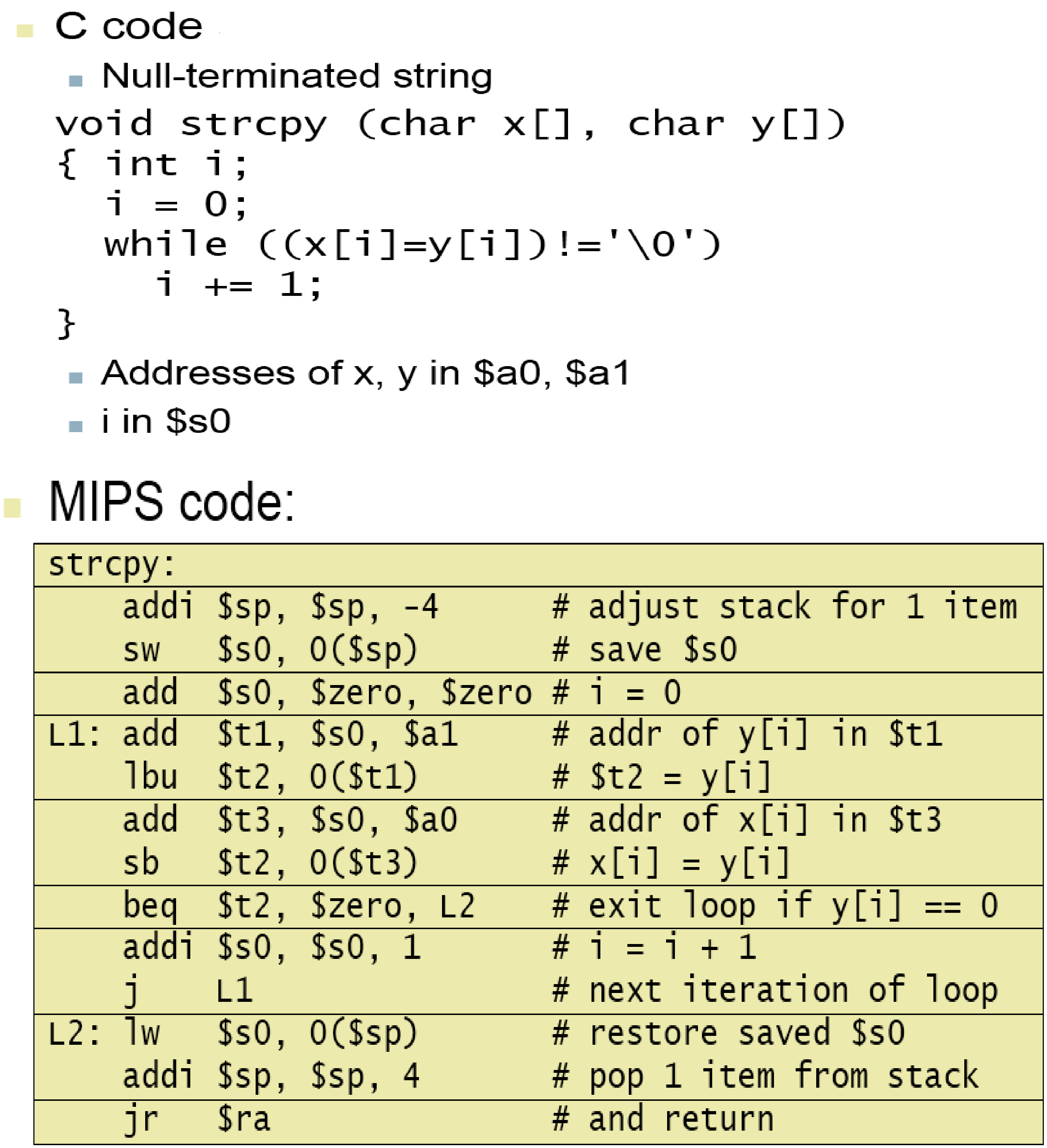


MIPS code:

Argument n in $a0, Result in $v0

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

String Copy Example



- C code
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in $a0, $a1
  - i in $s0

- MIPS code:

```
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw   $s0, 0($sp)       # save $s0
    add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1     # addr of y[i] in $t1
    lbu  $t2, 0($t1)       # $t2 = y[i]
    add  $t3, $s0, $a0     # addr of x[i] in $t3
    sb   $t2, 0($t3)       # x[i] = y[i]
    beq  $t2, $zero, L2    # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j    L1                # next iteration of loop
L2: lw   $s0, 0($sp)       # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr   $ra               # and return
```

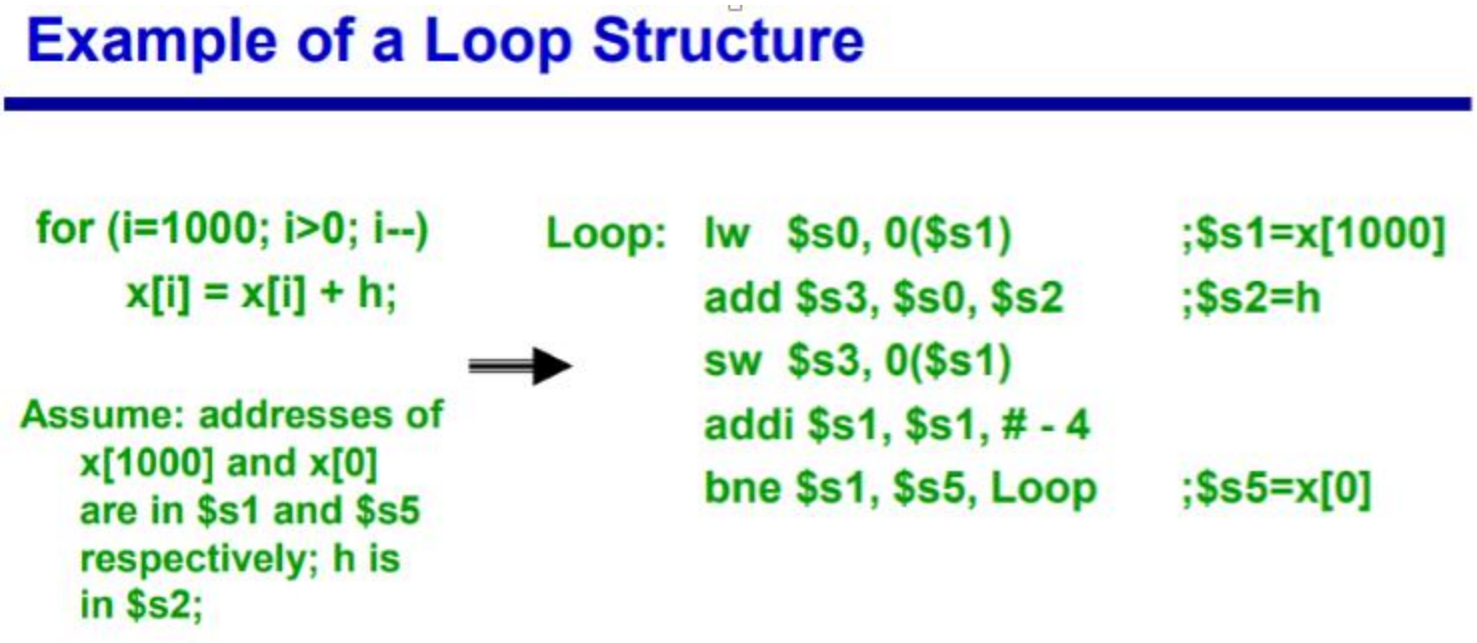
Copying a String

A string in C is an array of chars terminated with null char

```
i = 0;
do { ch = source[i]; target[i] = ch; i++; }
while (ch != '\0');
```

Given that: `$a0 = &target` and `$a1 = &source`

```
loop:
 lb    $t0, 0($a1)  # load byte: $t0 = source[i]
 sb    $t0, 0($a0)  # store byte: target[i]= $t0
 addiu $a0, $a0, 1  # $a0 = &target[i]
 addiu $a1, $a1, 1  # $a1 = &source[i]
 bnez  $t0, loop    # loop until NULL char
```



Example of a Loop Structure

```
for (i=1000; i>0; i--)
      x[i] = x[i] + h;
```

Assume: addresses of x[1000] and x[0] are in $s1 and $s5 respectively; h is in $s2;

```
Loop:  lw   $s0, 0($s1)          ;$s1=x[1000]
       add $s3, $s0, $s2         ;$s2=h
       sw  $s3, 0($s1)
       addi $s1, $s1, # - 4
       bne $s1, $s5, Loop        ;$s5=x[0]
```

MIPS Memory Layout

The top addresses from 0x80000000 to 0xFFFFFFFF are not available to user programs. They are used for the operating system and for ROM. When a MIPS chip is used in an embedded controller the control program exists in ROM in this upper half of the address space.

The parts of address space accessible to a user program are divided as follows:

Reserved

Memory below the text segment and above the stack is reserved for use by the operating system.

Text

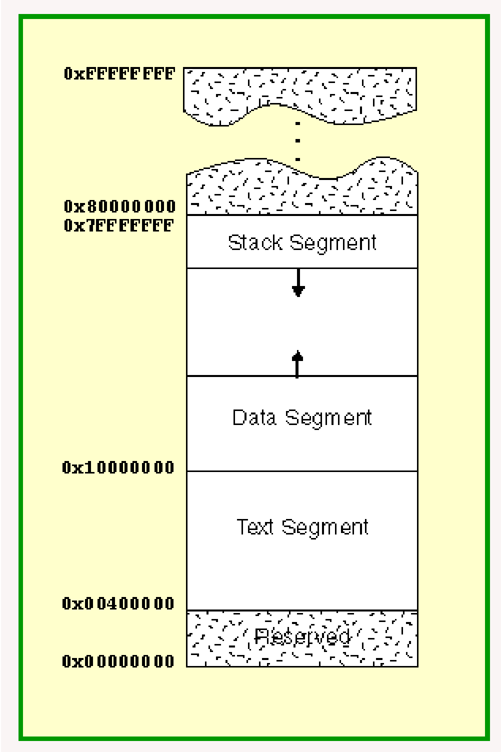
The assembly language instructions begin at address 0x400000.

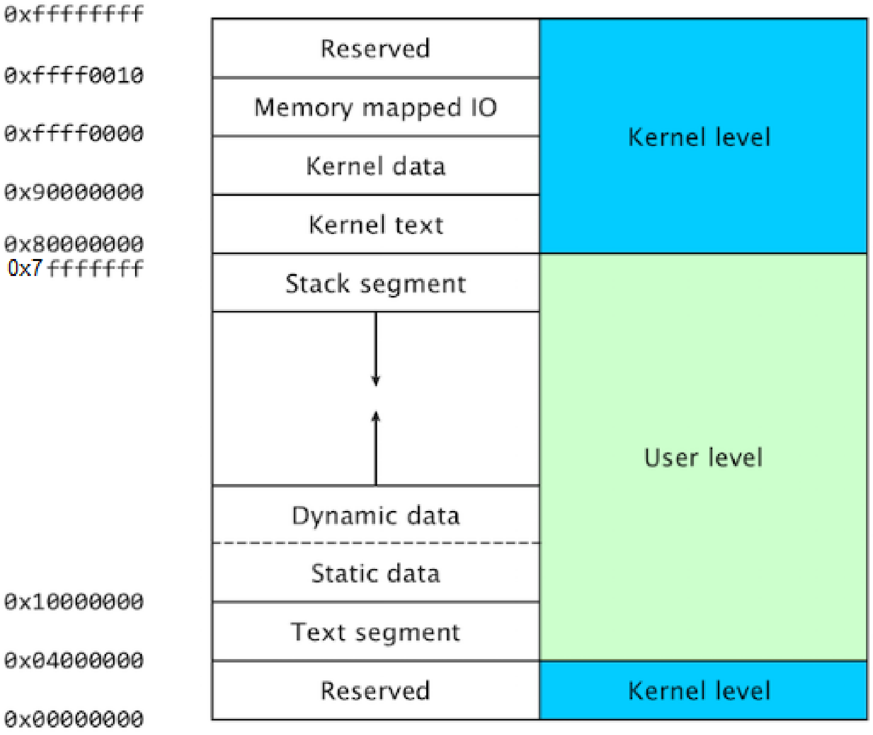
Data (Static + Dynamic)

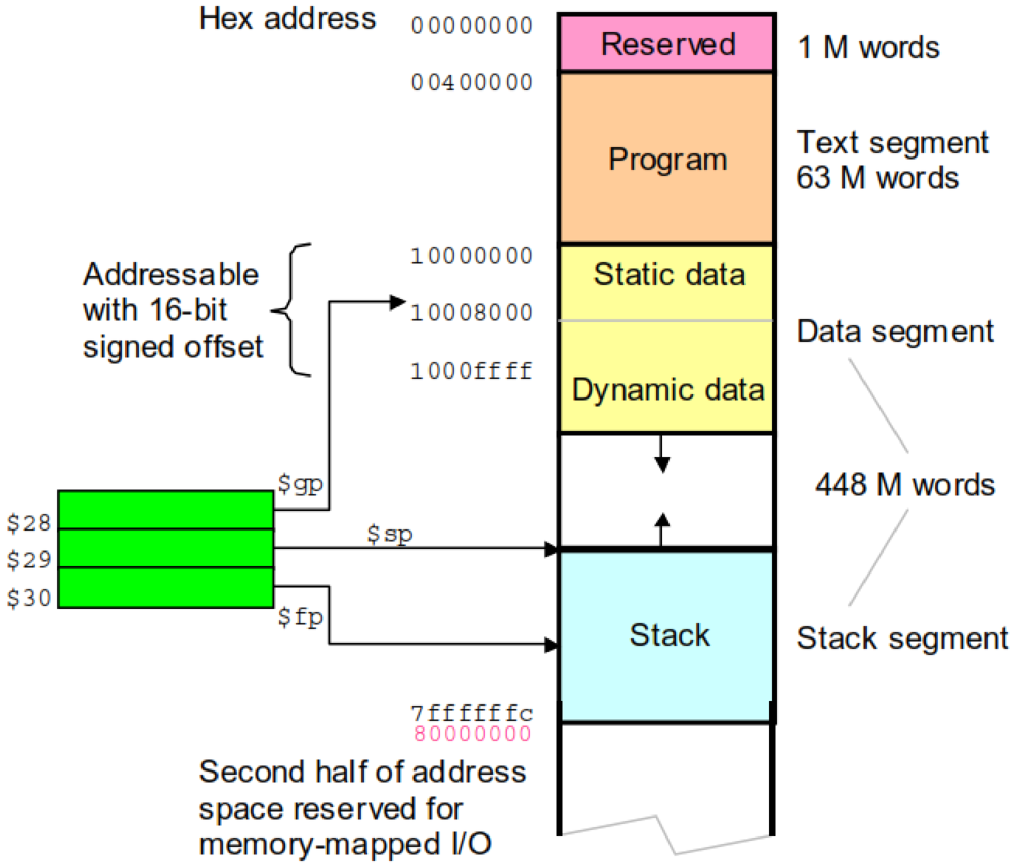
This holds the data that the program operates on. **Static data** store global variables (e.g., static variables in C, constant arrays and strings). $gp is reserved to point to static data. Dynamic data storage grows toward higher memory locations. **Dynamic data** is data that is allocated and deallocated as the program executes.

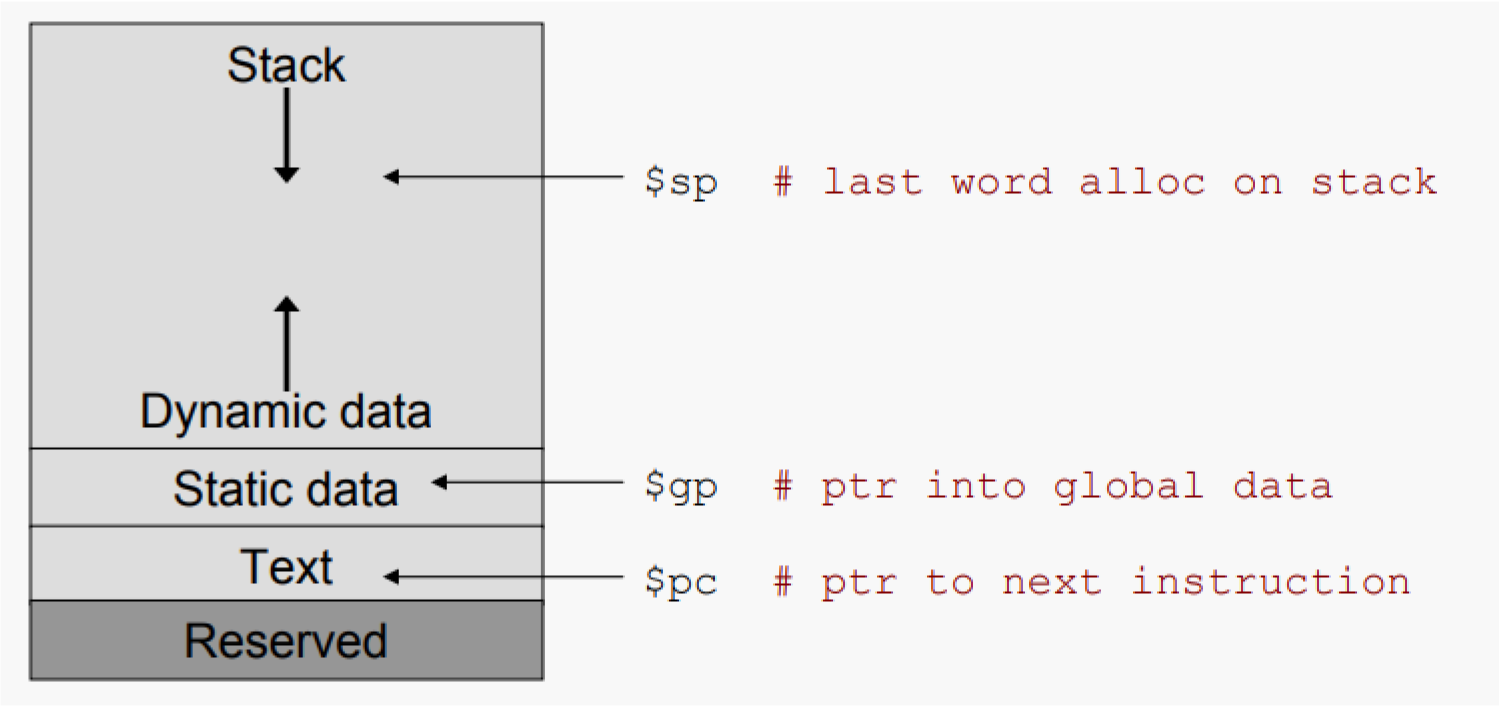
Stack

For function and procedure linkage. **$sp** is reserved to point to stack segment. The **$sp** is initialized to (7FFF FFFC)$_h$.

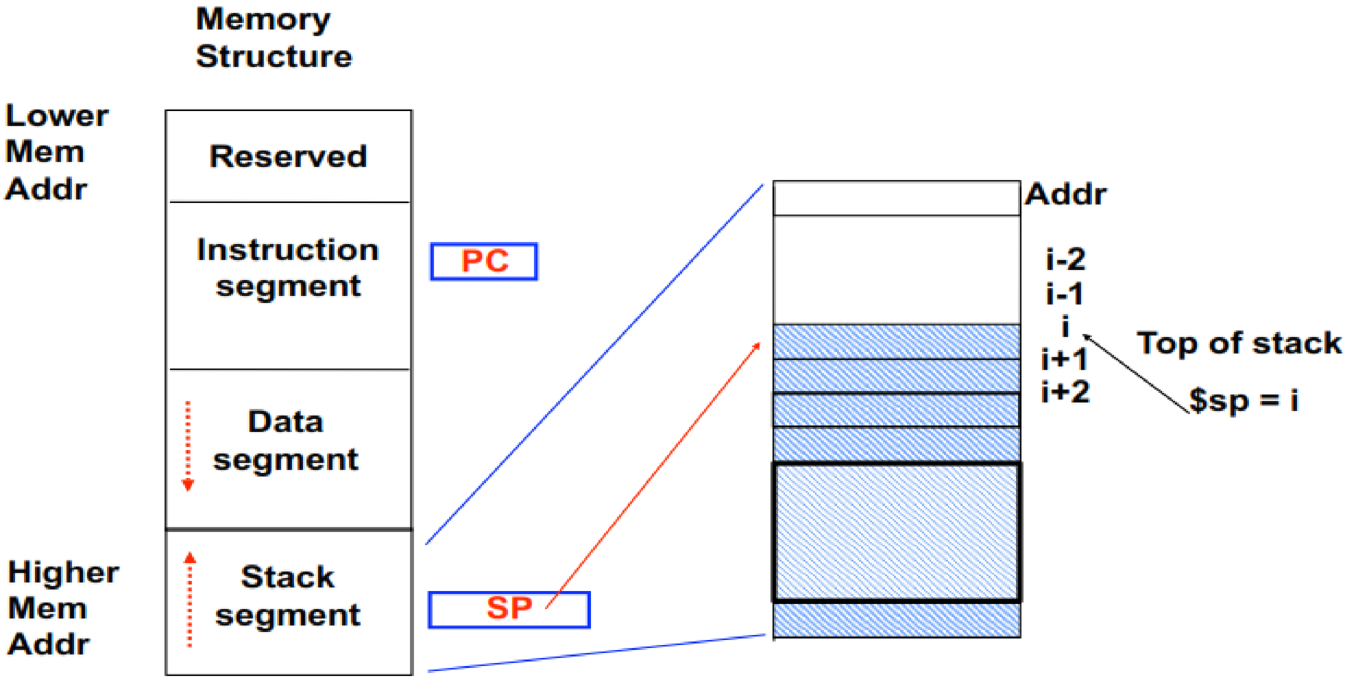





Hex address
00000000
Reserved
1 M words
00400000
Program
Text segment
63 M words
Addressable
with 16-bit
signed offset
10000000
Static data
10008000
1000ffff
Dynamic data
Data segment
$gp
448 M words
$28
$29
$30
$sp
$fp
Stack
Stack segment
7ffffffc
80000000
Second half of address
space reserved for
memory-mapped I/O

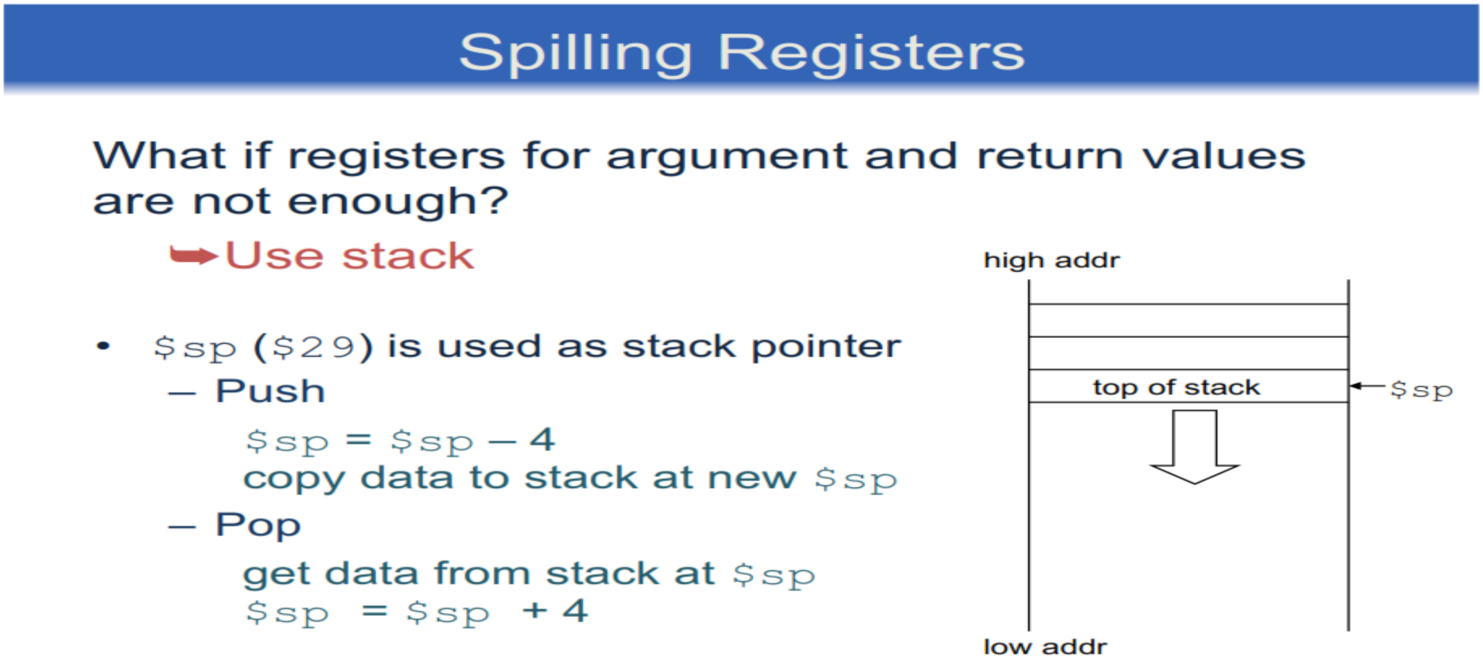

Stack
Dynamic data
Static data
Text
Reserved
$sp # last word alloc on stack
$gp # ptr into global data
$pc # ptr to next instruction

Where is the stack located?



- ❑ **Use of the Stack in procedure calls**
- ❑ **The stack**
  - ✓ A dedicated area (part) of the main memory.
  - ✓ LIFO
  - ✓ Hold values passed to a procedure as arguments
  - ✓ Save register contents when needed
  - ✓ Provide space for variables local to a procedure
  - ✓ Stack operations:
    - **sw** : place (push) data on stack
    - **lw** : remove (pop) data from stack
  - ✓ In MIPS, it grows from high address to low address as you push data on the stack.
  - ✓ Consequently, the content of the sp ($sp) decreases.
  - ✓ $29 ($sp) stores the address of the top of stack

Using the Stack for Data Storage



Spilling Registers

What if registers for argument and return values are not enough?

➥Use stack

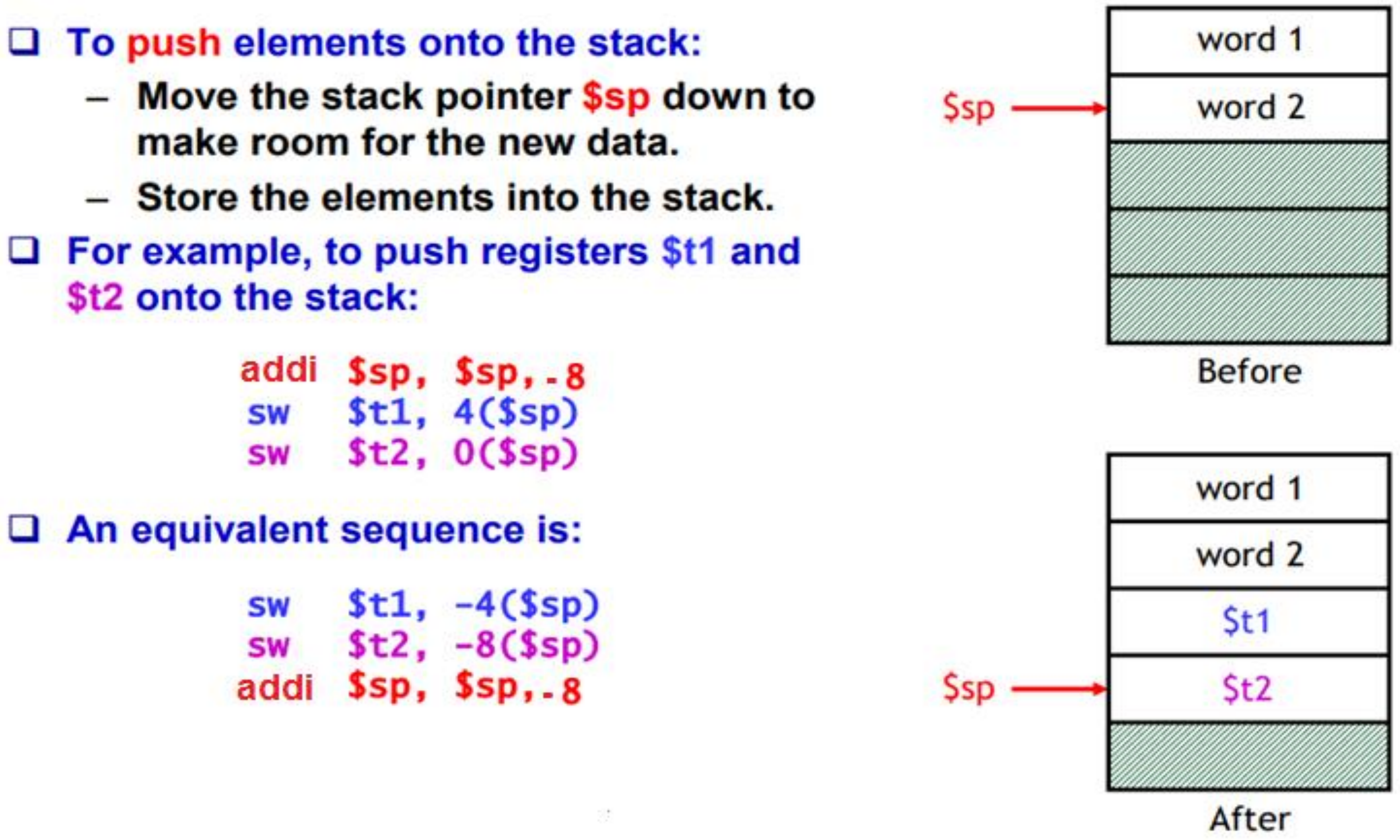
- $sp ($29) is used as stack pointer
  - Push

    $sp = $sp – 4

    copy data to stack at new $sp
  - Pop

    get data from stack at $sp

    $sp = $sp + 4



❑ To push elements onto the stack:

- Move the stack pointer $sp down to make room for the new data.
- Store the elements into the stack.

❑ For example, to push registers $t1 and $t2 onto the stack:

```
addi  $sp, $sp,-8
sw    $t1, 4($sp)
sw    $t2, 0($sp)
```

❑ An equivalent sequence is:

```
sw    $t1, -4($sp)
sw    $t2, -8($sp)
addi  $sp, $sp,-8
```

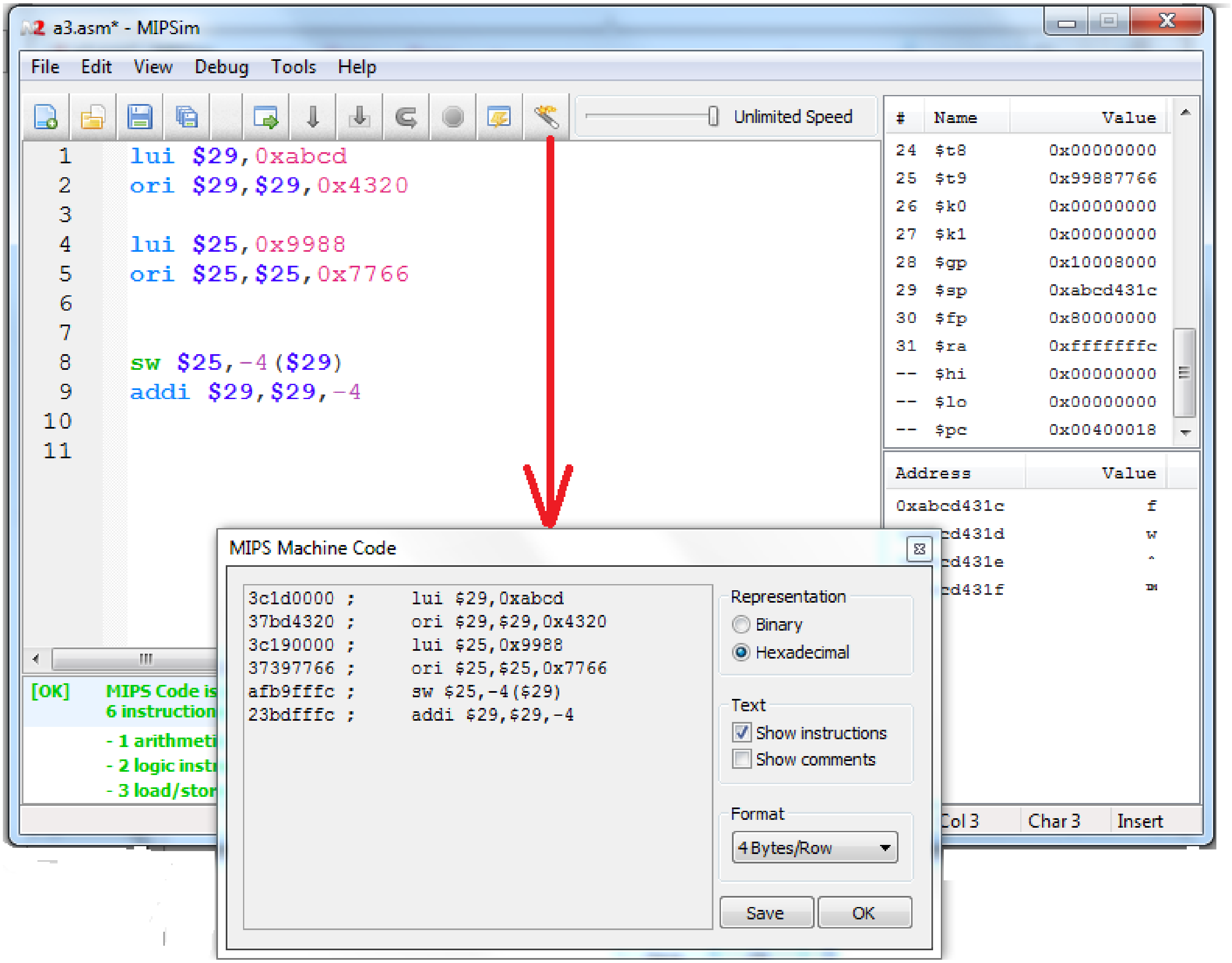
How Procedures use the Stack

- **But diffofsums overwrites 3 registers: $t0, $t1, $s0**

```
# MIPS assembly
# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr  $ra            # return to caller
```

Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12  # make space on stack
                      # to store 3 registers
  sw   $s0, 8($sp)    # save $s0 on stack
  sw   $t0, 4($sp)    # save $t0 on stack
  sw   $t1, 0($sp)    # save $t1 on stack
  add  $t0, $a0, $a1  # $t0 = f + g
  add  $t1, $a2, $a3  # $t1 = h + i
  sub  $s0, $t0, $t1  # result = (f + g) - (h + i)
  add  $v0, $s0, $0   # put return value in $v0
  lw   $t1, 0($sp)    # restore $t1 from stack
  lw   $t0, 4($sp)    # restore $t0 from stack
  lw   $s0, 8($sp)    # restore $s0 from stack
  addi $sp, $sp, 12   # deallocate stack space
  jr   $ra            # return to caller
```



```
lui $29,0xabcd
ori $29,$29,0x4320

lui $25,0x9988
ori $25,$25,0x7766


sw $25,-4($29)
addi $29,$29,-4
```

#	Name	Value
24	$t8	0x00000000
25	$t9	0x99887766
26	$k0	0x00000000
27	$k1	0x00000000
28	$gp	0x10008000
29	$sp	0xabcd431c
30	$fp	0x80000000
31	$ra	0xfffffffc
--	$hi	0x00000000
--	$lo	0x00000000
--	$pc	0x00400018

Address	Value
0xabcd431c	f
cd431d	w
cd431e	^
cd431f	™

[OK] MIPS Code is
6 instruction
- 1 arithmeti
- 2 logic instr
- 3 load/stor

Col 3 Char 3 Insert

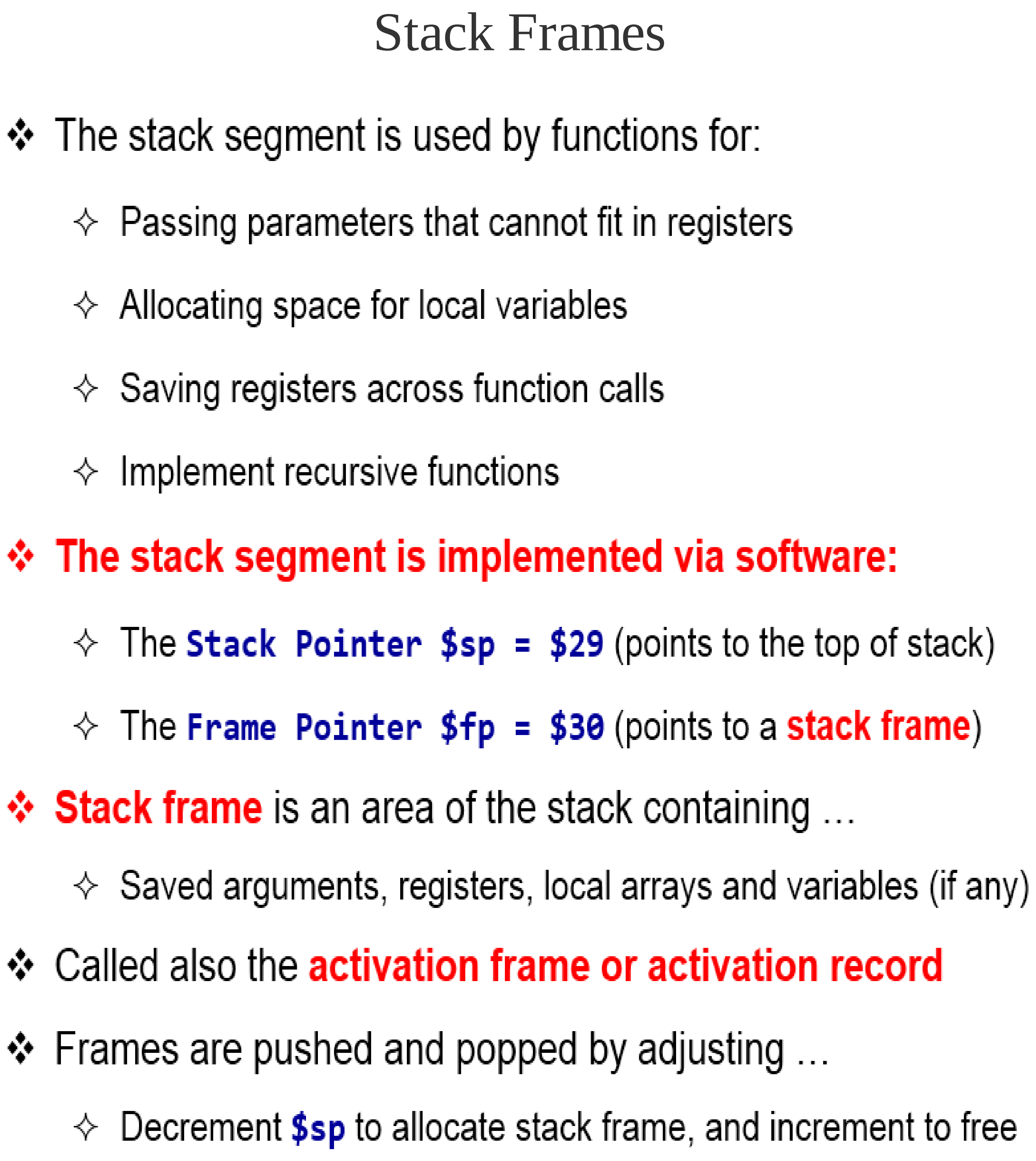
MIPS Machine Code

```
3c1d0000 ;      lui $29,0xabcd
37bd4320 ;      ori $29,$29,0x4320
3c190000 ;      lui $25,0x9988
37397766 ;      ori $25,$25,0x7766
afb9fffc ;      sw $25,-4($29)
23bdfffc ;      addi $29,$29,-4
```

Representation
○ Binary
● Hexadecimal

Text
☑ Show instructions
☐ Show comments

Format
4 Bytes/Row

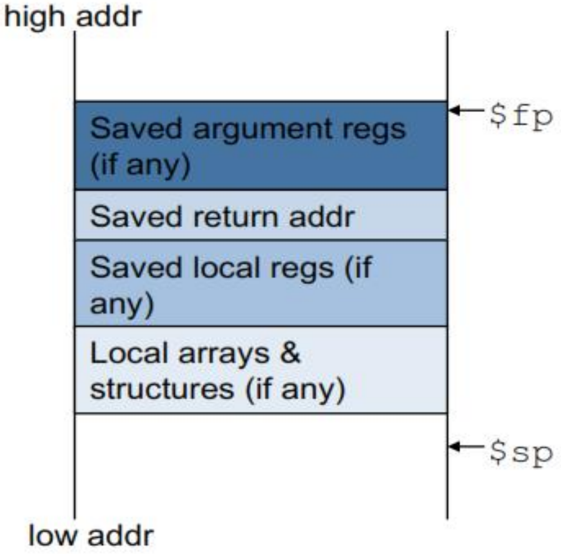


Stack Frames

- The stack segment is used by functions for:
  - Passing parameters that cannot fit in registers
  - Allocating space for local variables
  - Saving registers across function calls
  - Implement recursive functions
- **The stack segment is implemented via software:**
  - The **`Stack Pointer $sp = $29`** (points to the top of stack)
  - The **`Frame Pointer $fp = $30`** (points to a **stack frame**)
- **Stack frame** is an area of the stack containing …
  - Saved arguments, registers, local arrays and variables (if any)
- Called also the **activation frame or activation record**
- Frames are pushed and popped by adjusting …
  - Decrement **`$sp`** to allocate stack frame, and increment to free

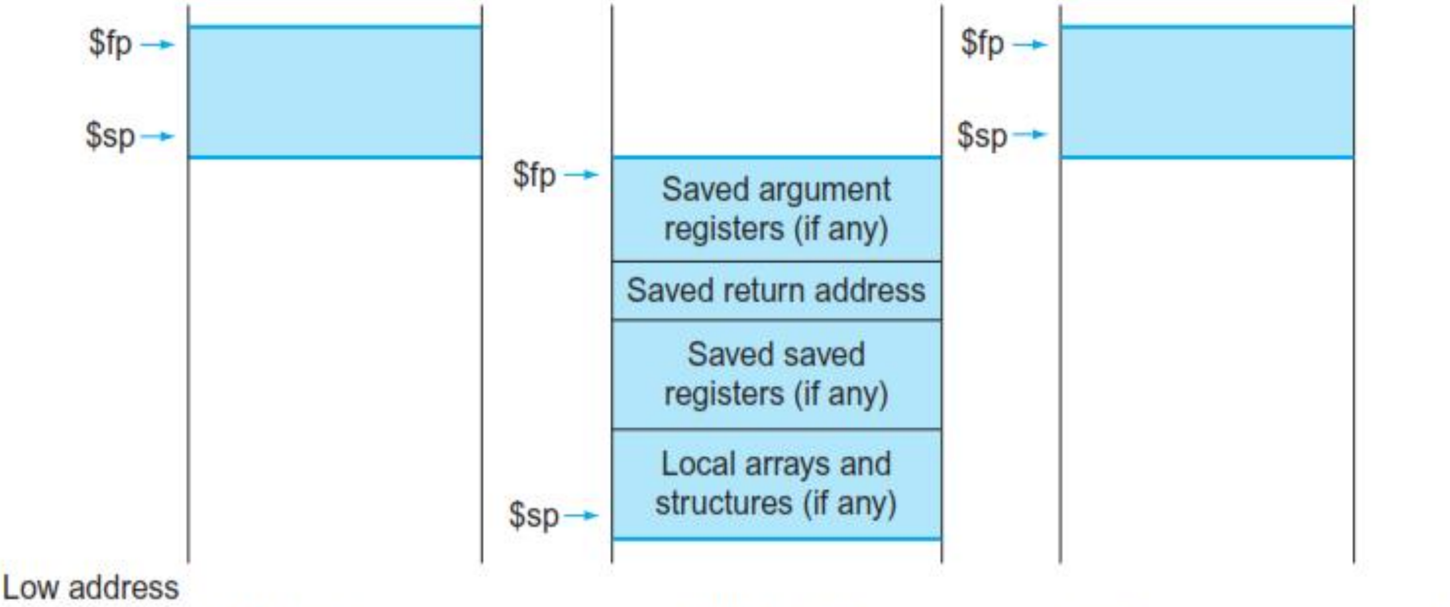
Allocating Space on the Stack

Procedure frame (aka activation record)
The segment of the stack containing a procedure's saved registers and local variables.


high addr
$fp
Saved argument regs (if any)
Saved return addr
Saved local regs (if any)
Local arrays & structures (if any)
$sp
low addr

The frame pointer (`$fp`) points to the first word of the frame of a procedure

- provides a stable "base" register for the procedure
- `$fp` is initialized using `$sp` on a call and `$sp` is restored using `$fp` on a return


$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address

(a) before, (b) during, (c) after the procedure call

MIPS Addressing Mode

The MIPS addressing modes are the following:

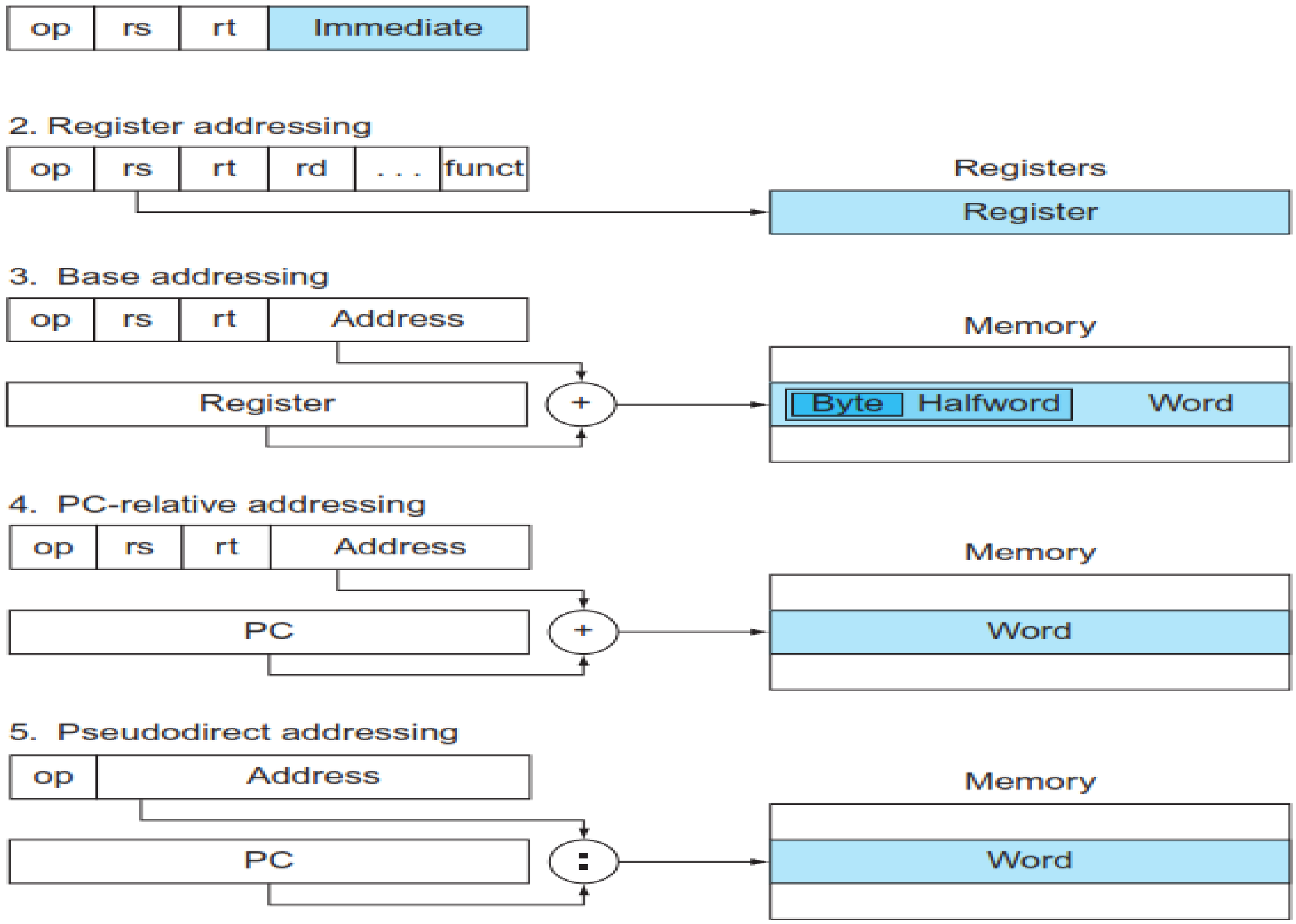
1. Immediate addressing, where the operand is a constant within the instruction itself

2. Register addressing, where the operand is a register.

3. Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction.

4. PC-relative addressing, where the branch address is the sum of the PC and a constant in the instruction.

5. Pseudodirect addressing, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC.



Register Only Addressing

- **Operands found in registers**
  - *Example:*
    `add $s0, $t2, $t3`
  - *Example:*
    `sub $t8, $s1, $0`

Immediate Addressing

- **16-bit immediate used as an operand**
  - *Example:*
    `addi $s4, $t5, -73`
  - *Example:*
    `ori  $t3, $t7, 0xFF`

Base Addressing

- **Address of operand is:**

  `base address + sign-extended immediate`

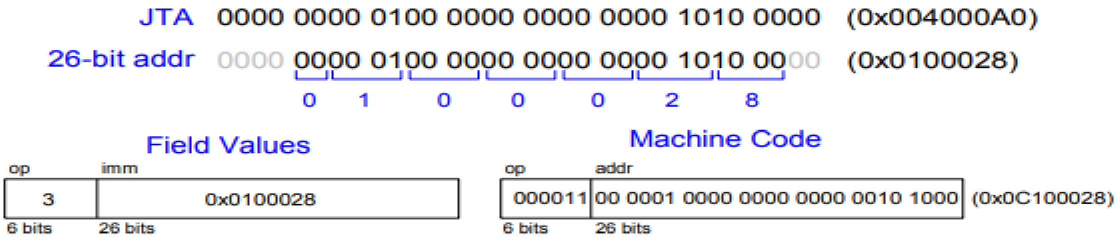
  - *Example:*
    `lw  $s4, 72($0)`   Address = `$0 + 72`
  - *Example:*
    `sw  $t2, -25($t1)`   Address = `$t1 - 25`

PC-Relative Addressing

```
0x10          beq     $t0, $0, else
0x14          addi    $v0, $0, 1
0x18          addi    $sp, $sp, i
0x1C          jr      $ra
0x20   else:  addi    $a0, $a0, -1
0x24          jal     factorial
```

Pseudo-direct Addressing

```
0x0040005C         jal      sum
...
0x004000A0   sum:  add      $v0, $a0, $a1
```



JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

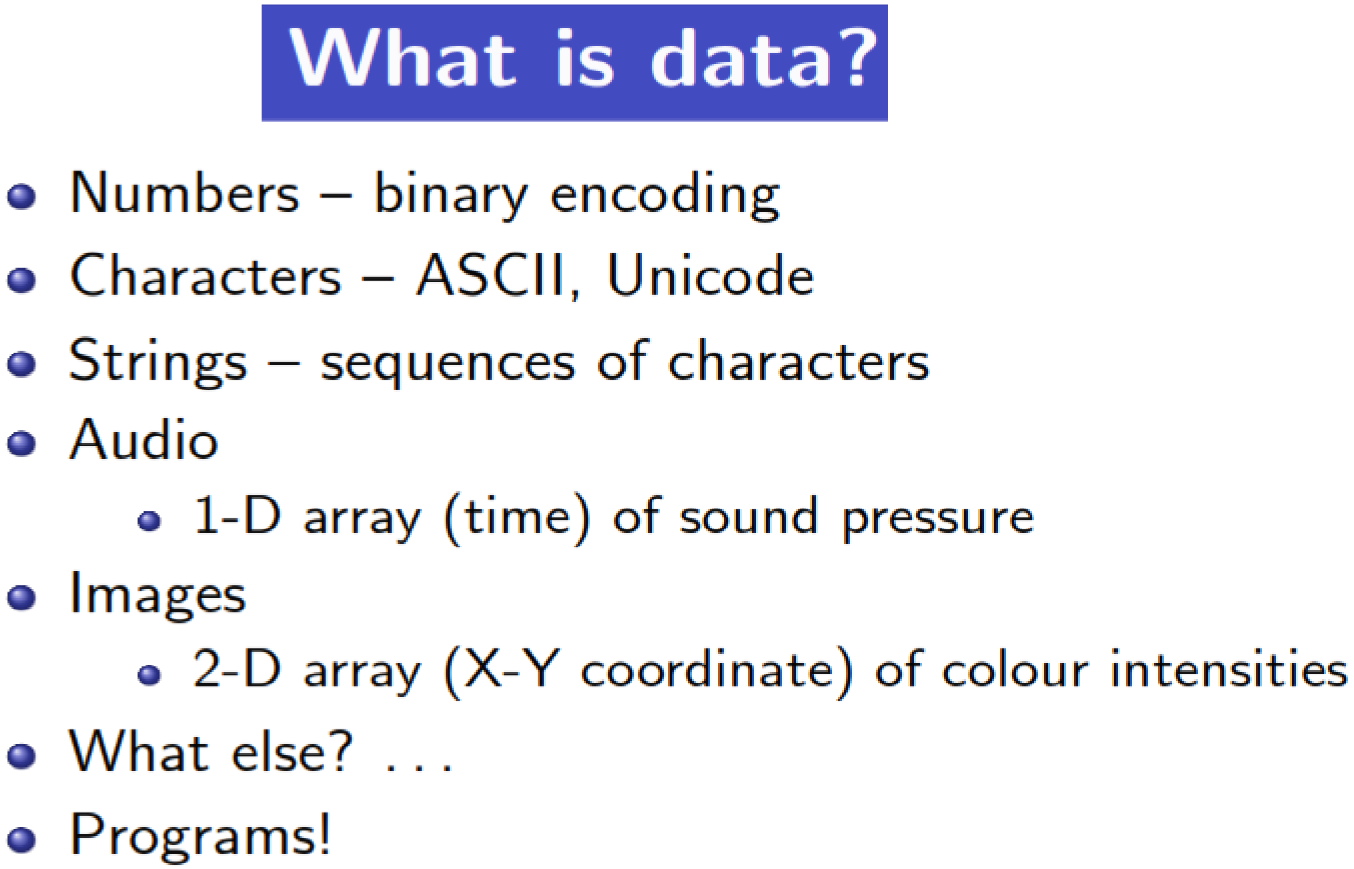
0 1 0 0 0 2 8

Field Values

op	imm
3	0x0100028
6 bits	26 bits

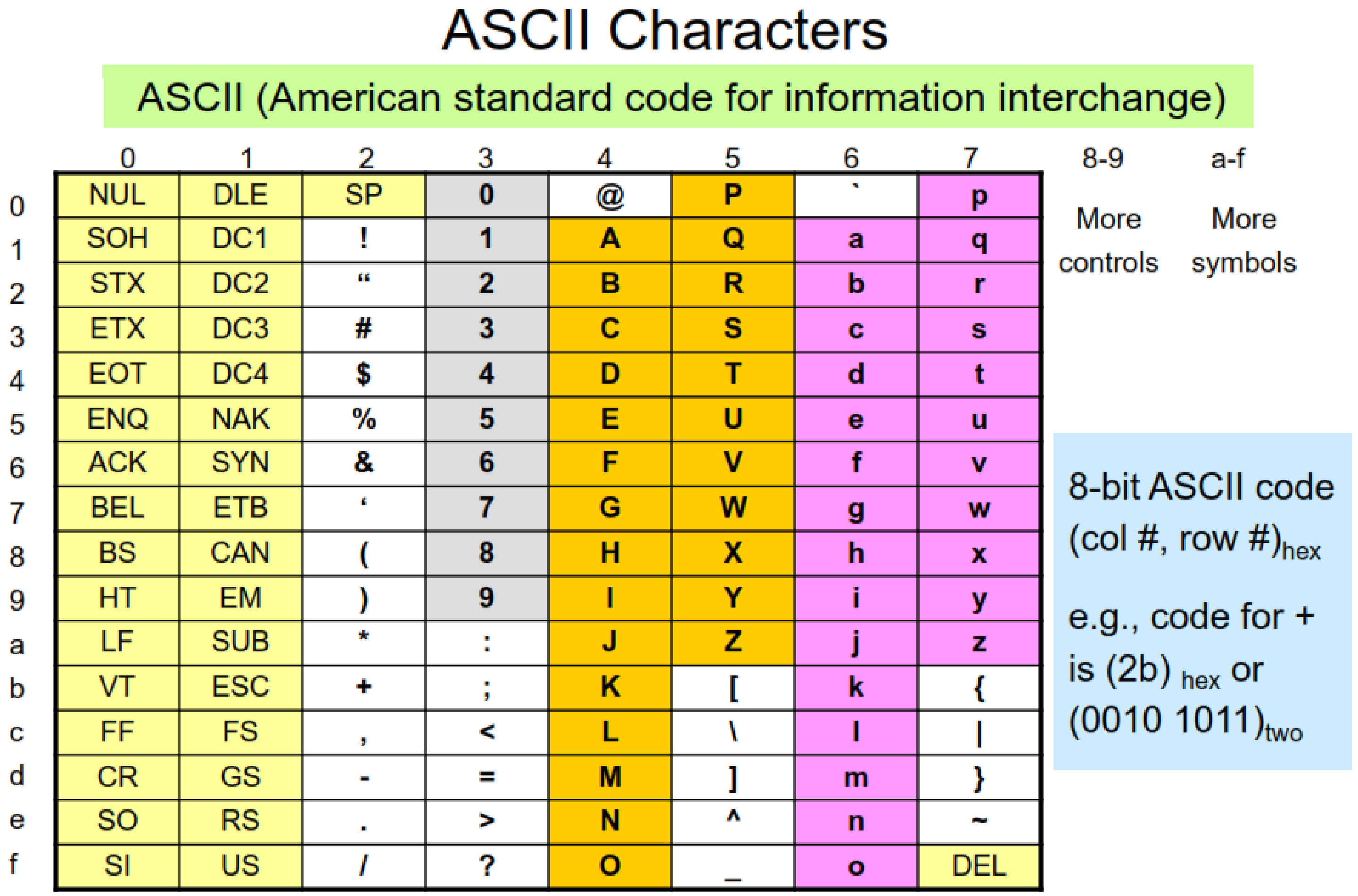
Machine Code

op	addr	
000011	00 0001 0000 0000 0000 0010 1000	(0x0C100028)
6 bits	26 bits	



What is data?

- Numbers – binary encoding
- Characters – ASCII, Unicode
- Strings – sequences of characters
- Audio
  - 1-D array (time) of sound pressure
- Images
  - 2-D array (X-Y coordinate) of colour intensities
- What else? . . .
- Programs!



ASCII Characters

ASCII (American standard code for information interchange)

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
a	LF	SUB	*	:	J	Z	j	z
b	VT	ESC	+	;	K	[	k	{
c	FF	FS	,	<	L	\	l	|
d	CR	GS	-	=	M	]	m	}
e	SO	RS	.	>	N	^	n	~
f	SI	US	/	?	O	_	o	DEL

8-9: More controls

a-f: More symbols

8-bit ASCII code (col #, row #)$_{hex}$

e.g., code for + is $(2b)_{hex}$ or $(0010\ 1011)_{two}$

Decoding Machine Code

- Decoding: Reverse-engineer machine language to create the assembly language
- Example: 00af 8020hex
  1. Convert hexadecimal to binary

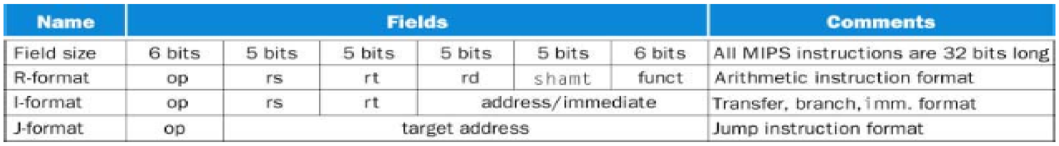
     0000 0000 1010 1111 1000 0000 0010 0000
  2. Look at the op field to determine the operation

     The op-field is 000000. It is an R-type instruction
  3. Decode the rest of the instruction by looking at the field values

```
op       rs      rt      rd      shamt   funct
000000   00101   01111   10000   00000   100000
```

  4. Reveal the assembly instruction

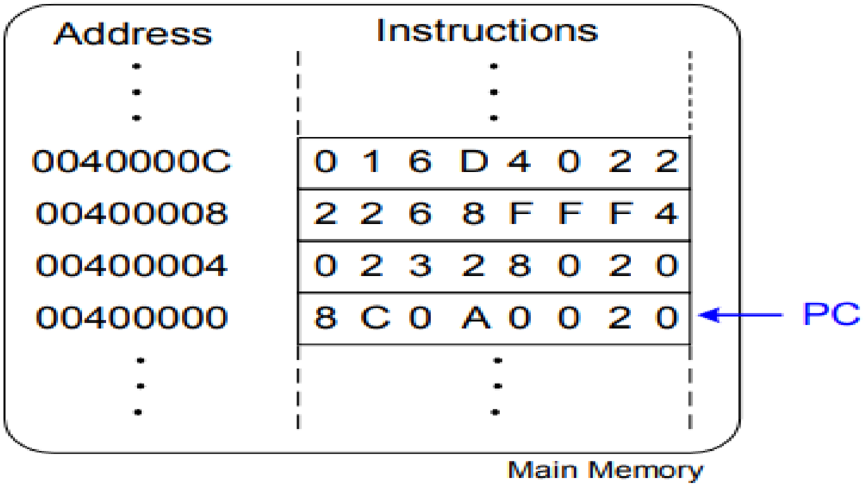
     add $s0, $a1, $t7

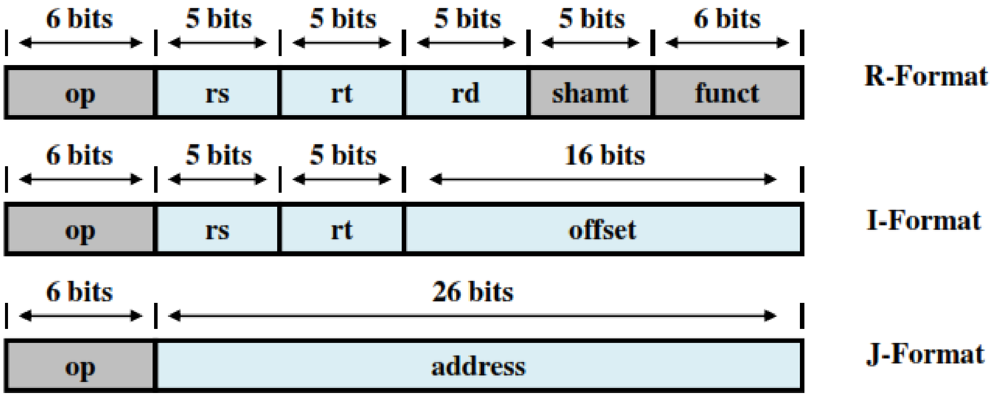


Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

```
Assembly Code              Machine Code
lw    $t2, 32($0)          0x8C0A0020
add   $s0, $s1, $s2        0x02328020
addi  $t0, $s3, -12        0x2268FFF4
sub   $t0, $t3, $t5        0x016D4022
```

Stored Program

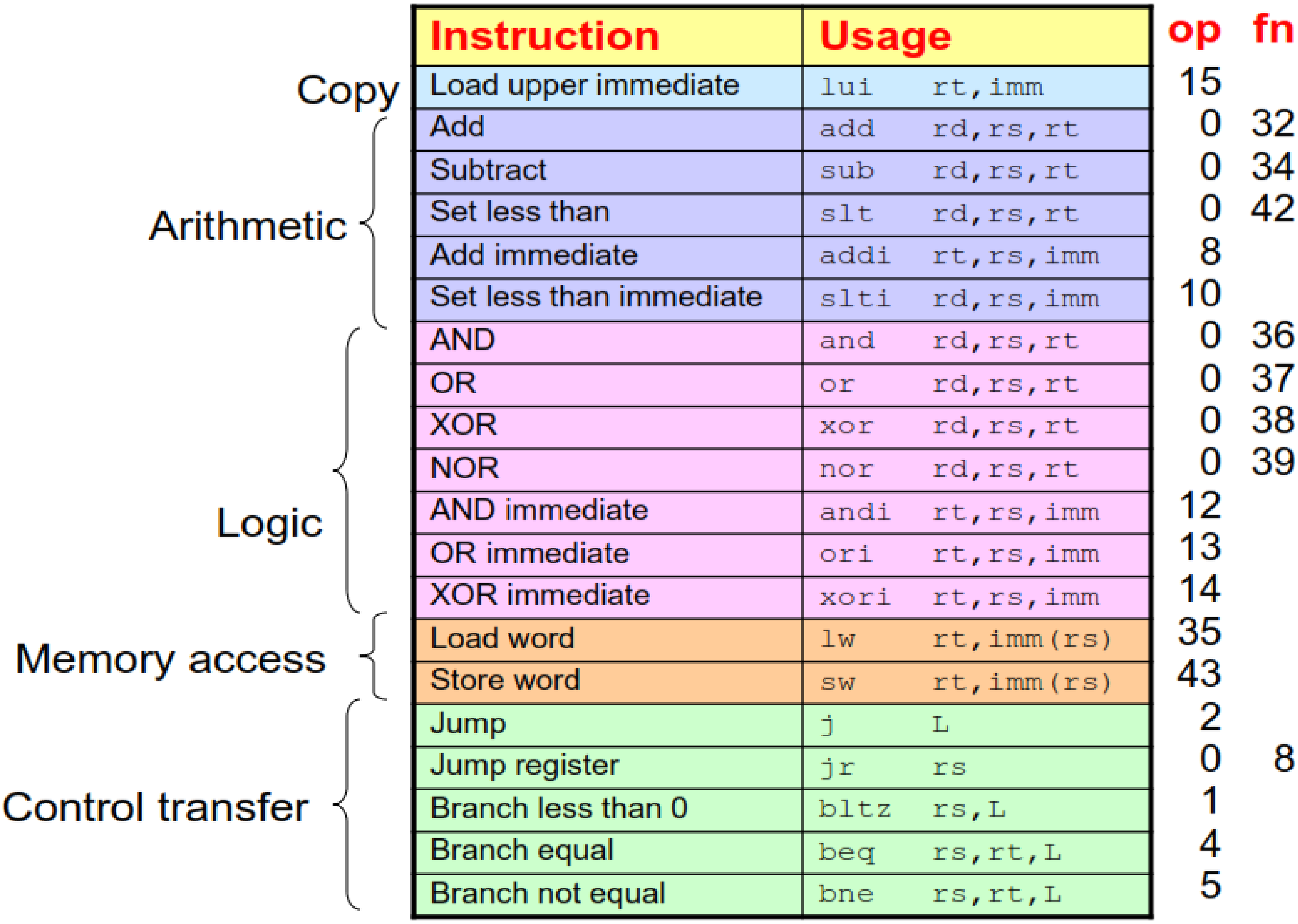




6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
op	rs	rt	rd	shamt	funct	R-Format

6 bits	5 bits	5 bits	16 bits	
op	rs	rt	offset	I-Format

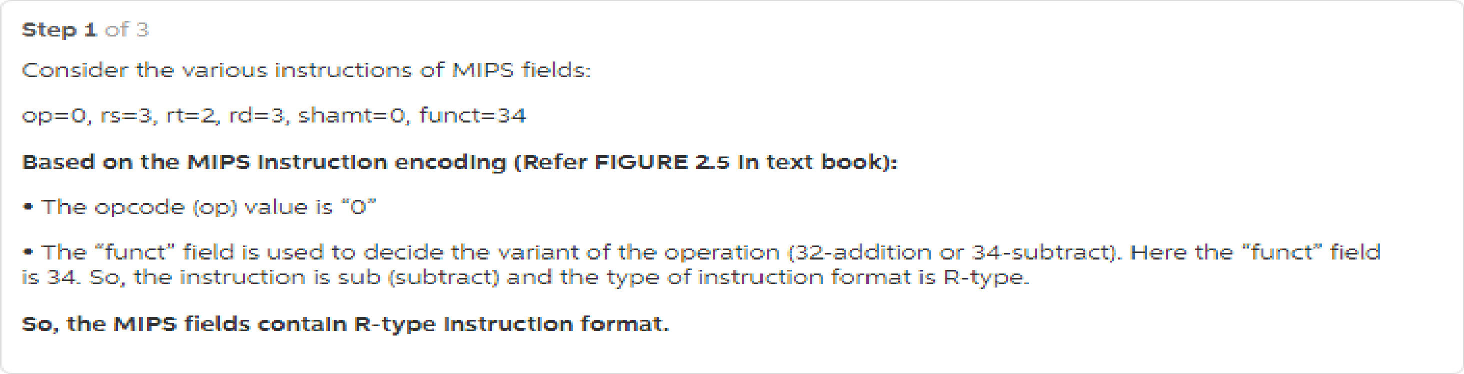
6 bits	26 bits	
op	address	J-Format



	Instruction	Usage	op	fn
Copy	Load upper immediate	lui rt,imm	15	
Arithmetic	Add	add rd,rs,rt	0	32
	Subtract	sub rd,rs,rt	0	34
	Set less than	slt rd,rs,rt	0	42
	Add immediate	addi rt,rs,imm	8	
	Set less than immediate	slti rd,rs,imm	10	
Logic	AND	and rd,rs,rt	0	36
	OR	or rd,rs,rt	0	37
	XOR	xor rd,rs,rt	0	38
	NOR	nor rd,rs,rt	0	39
	AND immediate	andi rt,rs,imm	12	
	OR immediate	ori rt,rs,imm	13	
	XOR immediate	xori rt,rs,imm	14	
Memory access	Load word	lw rt,imm(rs)	35	
	Store word	sw rt,imm(rs)	43	
Control transfer	Jump	j L	2	
	Jump register	jr rs	0	8
	Branch less than 0	bltz rs,L	1	
	Branch equal	beq rs,rt,L	4	
	Branch not equal	bne rs,rt,L	5	

2.16) Provide the type, assembly language instruction, and binary representation of instruction described by the following MIPS fields:

op=0 rs=3 rt=2 rd=3 shamt=0 funct=34



Step 1 of 3

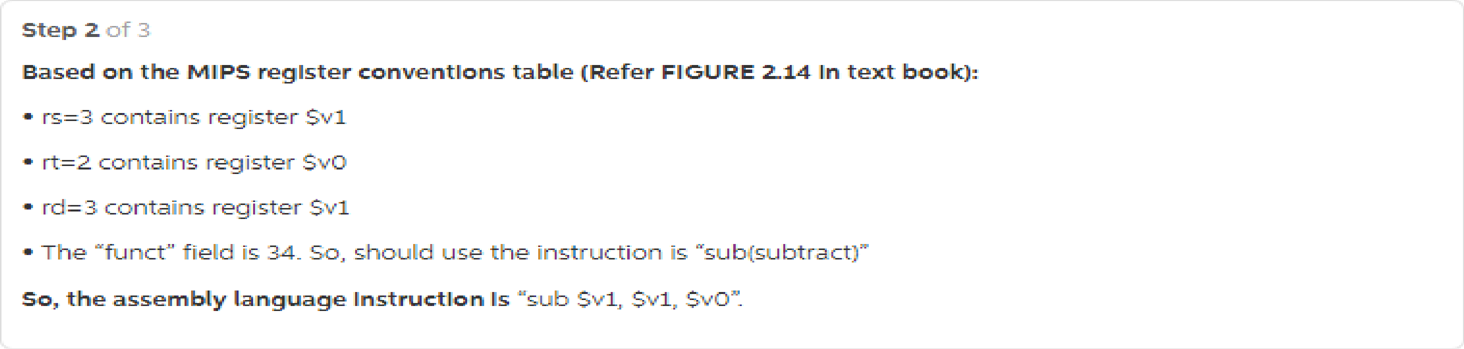
Consider the various instructions of MIPS fields:

op=0, rs=3, rt=2, rd=3, shamt=0, funct=34

Based on the MIPS instruction encoding (Refer FIGURE 2.5 in text book):

- The opcode (op) value is "0"
- The "funct" field is used to decide the variant of the operation (32-addition or 34-subtract). Here the "funct" field is 34. So, the instruction is sub (subtract) and the type of instruction format is R-type.

So, the MIPS fields contain R-type instruction format.

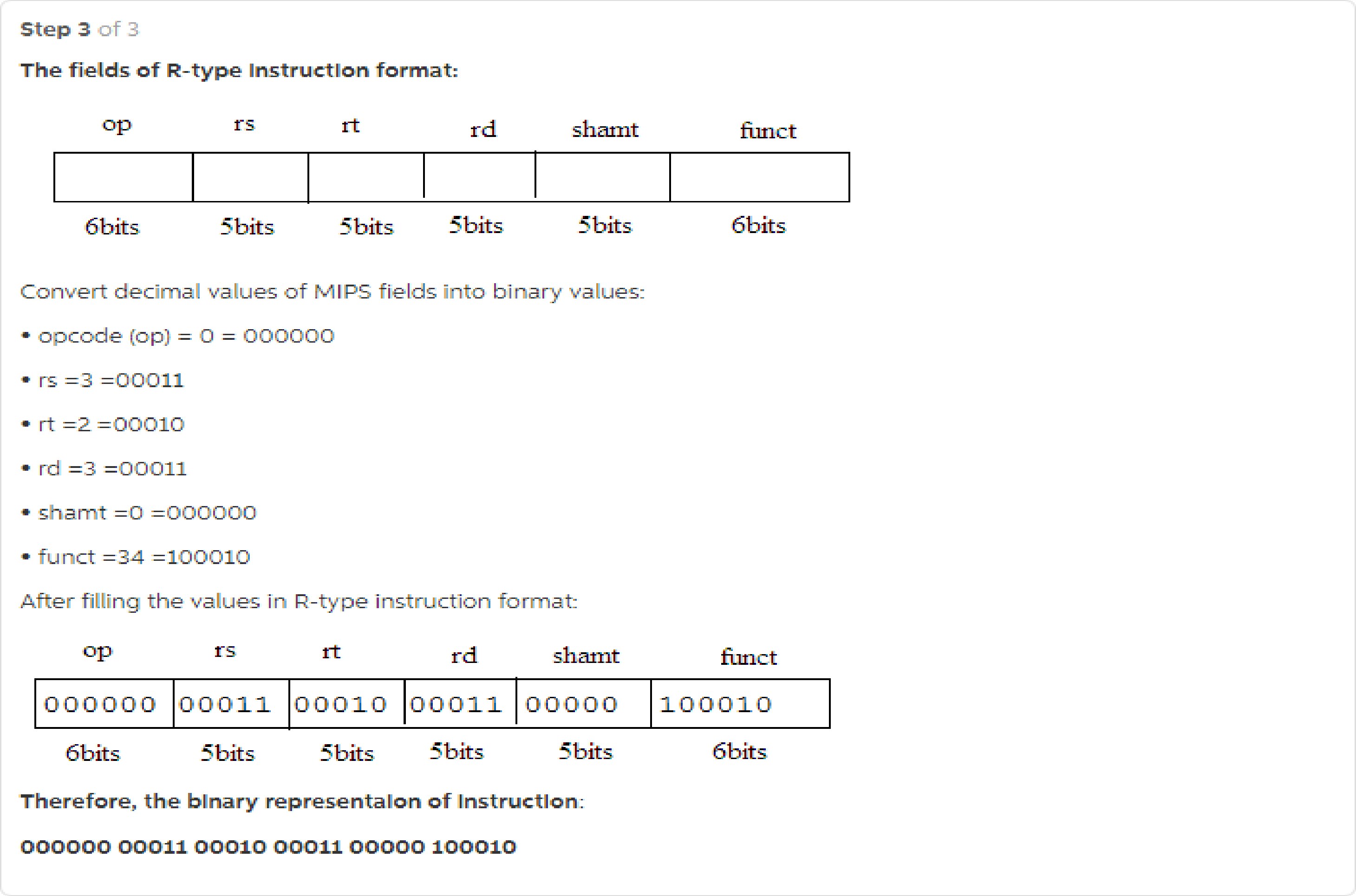


Step 2 of 3

Based on the MIPS register conventions table (Refer FIGURE 2.14 in text book):

- rs=3 contains register $v1
- rt=2 contains register $v0
- rd=3 contains register $v1
- The "funct" field is 34. So, should use the instruction is "sub(subtract)"

So, the assembly language instruction is "sub $v1, $v1, $v0".



Step 3 of 3

The fields of R-type instruction format:

op	rs	rt	rd	shamt	funct
6bits	5bits	5bits	5bits	5bits	6bits

Convert decimal values of MIPS fields into binary values:

- opcode (op) = 0 = 000000
- rs =3 =00011
- rt =2 =00010
- rd =3 =00011
- shamt =0 =000000
- funct =34 =100010

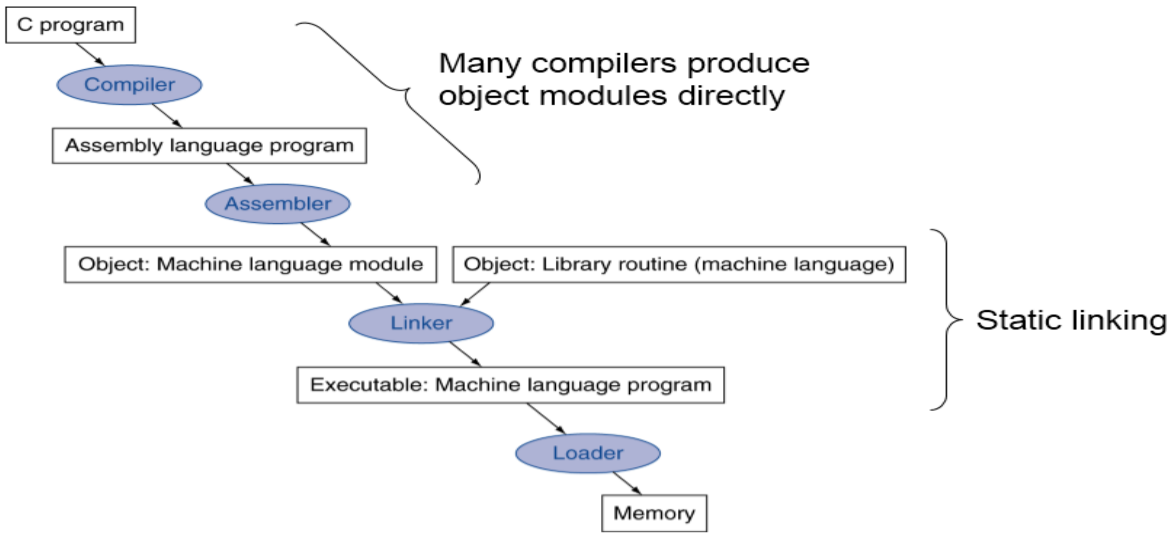
After filling the values in R-type instruction format:

op	rs	rt	rd	shamt	funct
000000	00011	00010	00011	00000	100010
6bits	5bits	5bits	5bits	5bits	6bits

Therefore, the binary representaion of instruction:

000000 00011 00010 00011 00000 100010

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

The linker has the following responsibilities:

- Ensuring correct interpretation (resolution) of labels in all modules
- Determining the placement of text and data segments in memory
- Evaluating all data addresses and instruction labels

The loader is in charge of the following:

- Determining the memory needs of the program from its header
- Copying text and data from the executable program file into memory
- Modifying (shifting) addresses, where needed, during copying
- Placing program parameters onto the stack (as in a procedure call)
- Initializing all machine registers, including the stack pointer
- Jumping to a start-up routine that calls the program's main routine

Program Template

```
# Title:                        Filename:
# Author:                       Date:
# Description:
# Input:
# Output:
################# Data segment #####################
.data
 . . .
################# Code segment #####################
.text
.globl main
main:                           # main program entry
 . . .
li $v0, 10                      # Exit program
syscall
```

- **.DATA** directive
  - Defines the data segment of a program containing data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables
- **.TEXT** directive
  - Defines the code segment of a program containing instructions
- **.GLOBL** directive
  - Declares a symbol as global
  - Global symbols can be referenced from other files
  - We use this directive to declare *main* procedure of a program

Data Directives

- **.BYTE** Directive
  - Stores the list of values as 8-bit bytes
- **.HALF** Directive
  - Stores the list as 16-bit values aligned on half-word boundary
- **.WORD** Directive
  - Stores the list as 32-bit values aligned on a word boundary
- **.WORD w:n** Directive
  - Stores the 32-bit value *w* into *n* consecutive words aligned on a word boundary.
- **.HALF w:n** Directive
  - Stores the 16-bit value *w* into *n* consecutive half-words aligned on a half-word boundary .
- **.BYTE w:n** Directive
  - Stores the 8-bit value *w* into *n* consecutive bytes.
- **.FLOAT** Directive
  - Stores the listed values as single-precision floating point
- **.DOUBLE** Directive
  - Stores the listed values as double-precision floating point

String Directives

(No alignment is performed)

- **.ASCII** Directive
  - Allocates a sequence of bytes for an ASCII string
- **.ASCIIZ** Directive
  - Same as **.ASCII** directive, but adds a NULL char at end of string
  - Strings are null-terminated, as in the C programming language
- **.SPACE n** Directive
  - Allocates space of *n* uninitialized bytes in the data segment
- Special characters in strings follow C convention
  - Newline: \n   Tab:\t   Quote: \”

Examples of Data Definitions

```
.DATA
var1:   .BYTE     'A', 'E', 127, -1, '\n'
var2:   .HALF     -10, 0xffff
var3:   .WORD     0x12345678
Var4:   .WORD     0:10
var5:   .FLOAT    12.3, -0.1
var6:   .DOUBLE   1.5e-10
str1:   .ASCII    "A String\n"
array:  .SPACE    100
```

Memory Alignment

- Memory is viewed as an array of bytes with addresses
  - Byte Addressing: address points to a byte in memory
- Words occupy 4 consecutive bytes in memory
  - MIPS instructions and integers occupy 4 bytes
- Alignment: address is a multiple of size
  - Word address should be a multiple of 4
    - Least significant 2 bits of address should be 00
  - Halfword address should be a multiple of 2
- `.ALIGN n` directive
  - Aligns the next data definition on a 2^n byte boundary

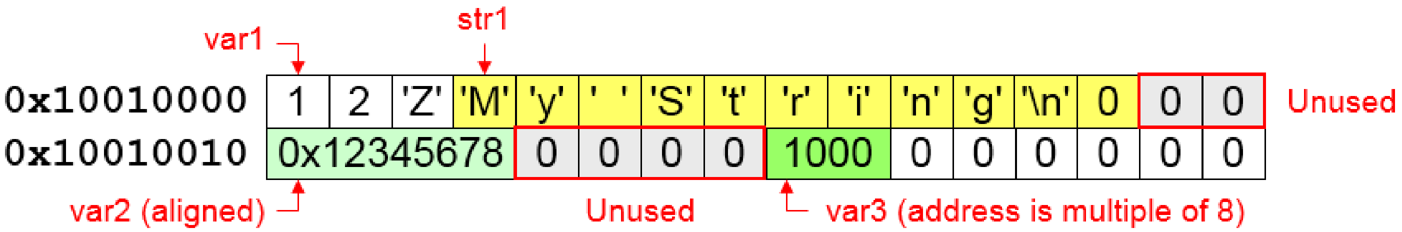
Symbol Table

- Assembler builds a symbol table for labels (variables)
  - Assembler computes the address of each label in data segment
- Example

```
.DATA
var1:   .BYTE    1, 2,'Z'
str1:   .ASCIIZ "My String\n"
var2:   .WORD    0x12345678
.ALIGN  3
var3:   .HALF    1000
```

Symbol Table

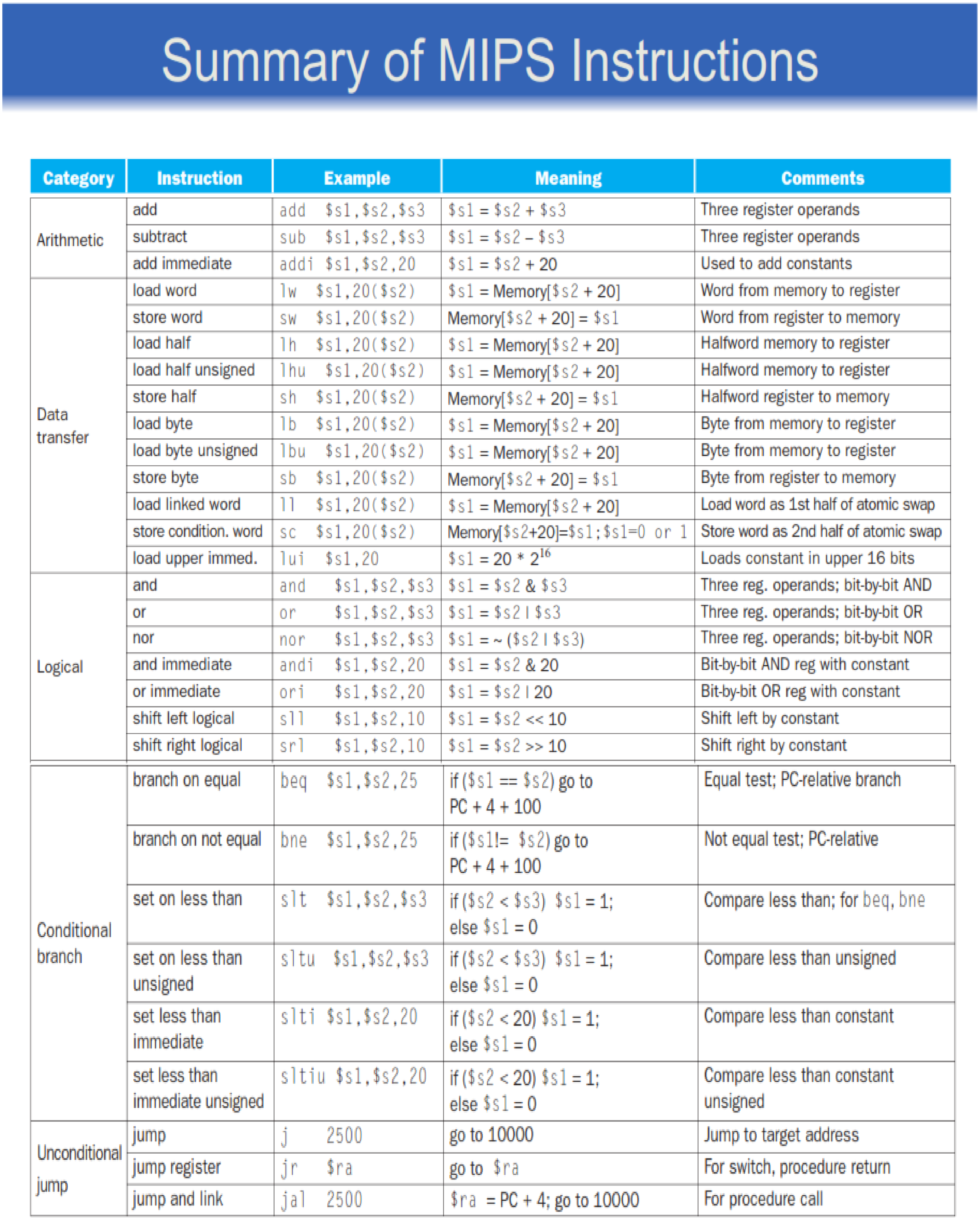
Label	Address
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018



var1 str1

0x10010000	1	2	'Z'	'M'	'y'	' '	'S'	't'	'r'	'i'	'n'	'g'	'\n'	0	0	0	Unused
0x10010010	0x12345678				0	0	0	0	1000		0	0	0	0	0	0	

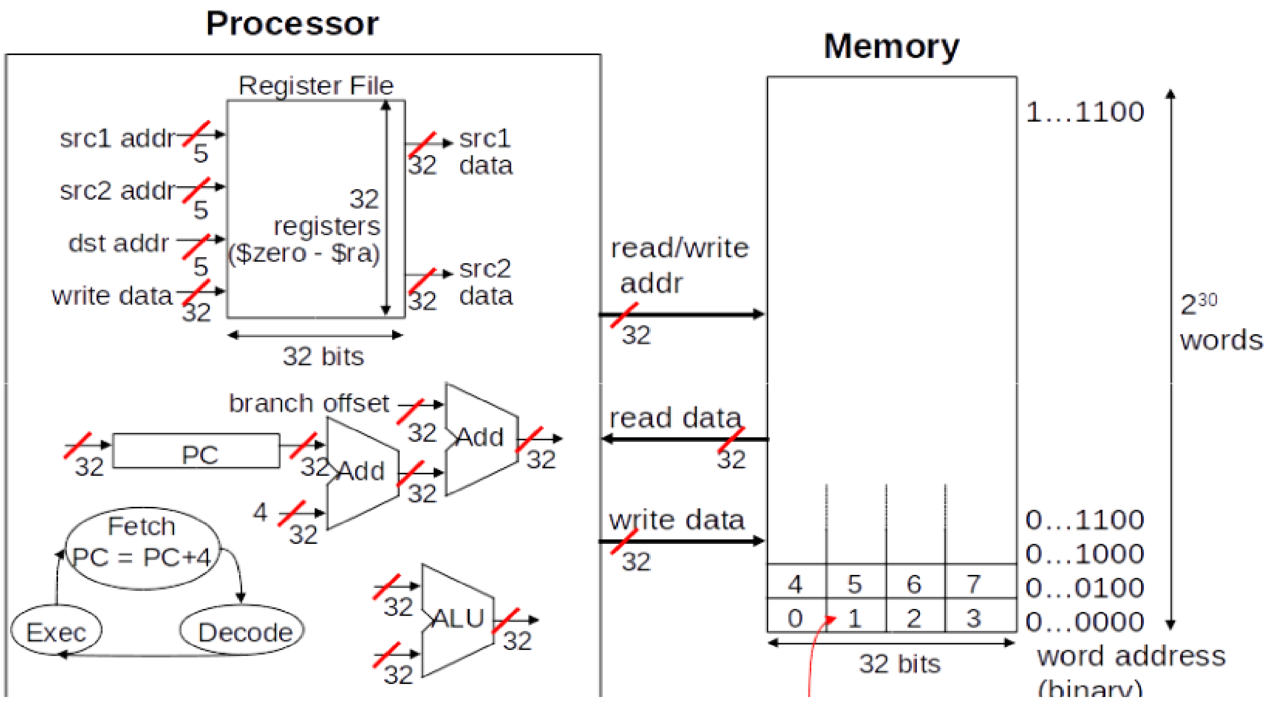
var2 (aligned) Unused var3 (address is multiple of 8)



Summary of MIPS Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

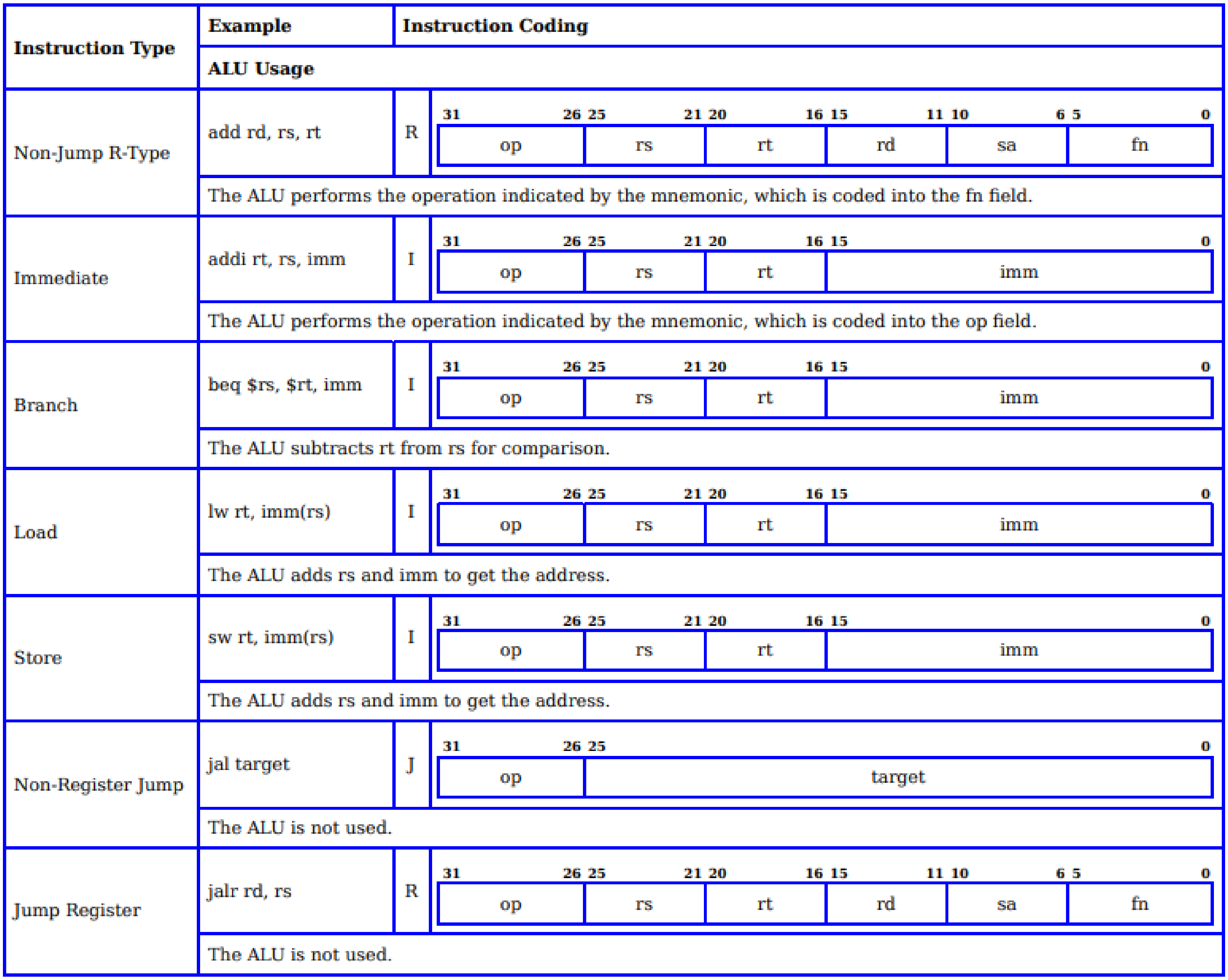
MIPS Organization Summary



MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions - 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits
- Good design demands good compromises
  - 3 basic instruction formats
- Smaller is faster
  - limited instruction set
  - limited number (32) of registers in register file
  - limited number (5) of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

MIPS Instruction Implementation Types



Instruction Type	Example		Instruction Coding
	ALU Usage		
Non-Jump R-Type	add rd, rs, rt	R	31 26 25 21 20 16 15 11 10 6 5 0 op rs rt rd sa fn
	The ALU performs the operation indicated by the mnemonic, which is coded into the fn field.		
Immediate	addi rt, rs, imm	I	31 26 25 21 20 16 15 0 op rs rt imm
	The ALU performs the operation indicated by the mnemonic, which is coded into the op field.		
Branch	beq $rs, $rt, imm	I	31 26 25 21 20 16 15 0 op rs rt imm
	The ALU subtracts rt from rs for comparison.		
Load	lw rt, imm(rs)	I	31 26 25 21 20 16 15 0 op rs rt imm
	The ALU adds rs and imm to get the address.		
Store	sw rt, imm(rs)	I	31 26 25 21 20 16 15 0 op rs rt imm
	The ALU adds rs and imm to get the address.		
Non-Register Jump	jal target	J	31 26 25 0 op target
	The ALU is not used.		
Jump Register	jalr rd, rs	R	31 26 25 21 20 16 15 11 10 6 5 0 op rs rt rd sa fn
	The ALU is not used.		

Homework #3

- **Exercises in the Textbook (Computer Organization & Design, by Patterson & Hennessy, 5th Edition).**
  - **2.1, 2.3, 2.4, 2.6, 2.7, 2.10, 2.16, 2.23 , 2.27and 2.38.**
- **LAB (MIPSim2)**
  - **2.12 and 2.19**

Chapter 3
Arithmetic for Computers

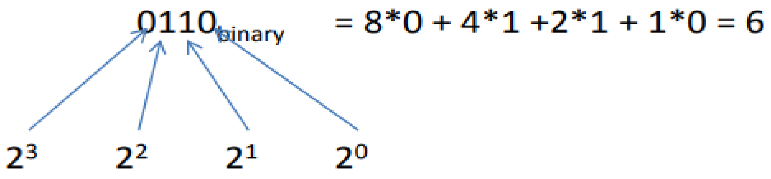
- **Operations on integers**
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- **Floating-point real numbers**
  - Representation and operations

Fixed-radix positional representation with *k* digits

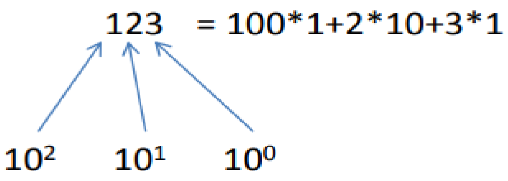
Value of a number: $x = (x_{k-1}x_{k-2}\ldots x_1x_0)_r = \sum_{i=0}^{k-1} x_i r^i$

For example:

$27 = (11011)_{two} = (1\times2^4) + (1\times2^3) + (0\times2^2) + (1\times2^1) + (1\times2^0)$

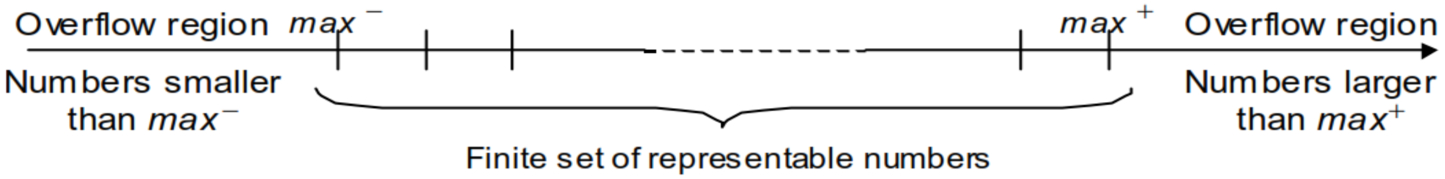


0110_{binary} = 8*0 + 4*1 +2*1 + 1*0 = 6



123 = 100*1+2*10+3*1

Representation Range and Overflow



Overflow region
max^-
max^+
Overflow region
Numbers smaller than max^-
Numbers larger than max^+
Finite set of representable numbers

Note:-

- an unsigned integer containing n bits can have a value between

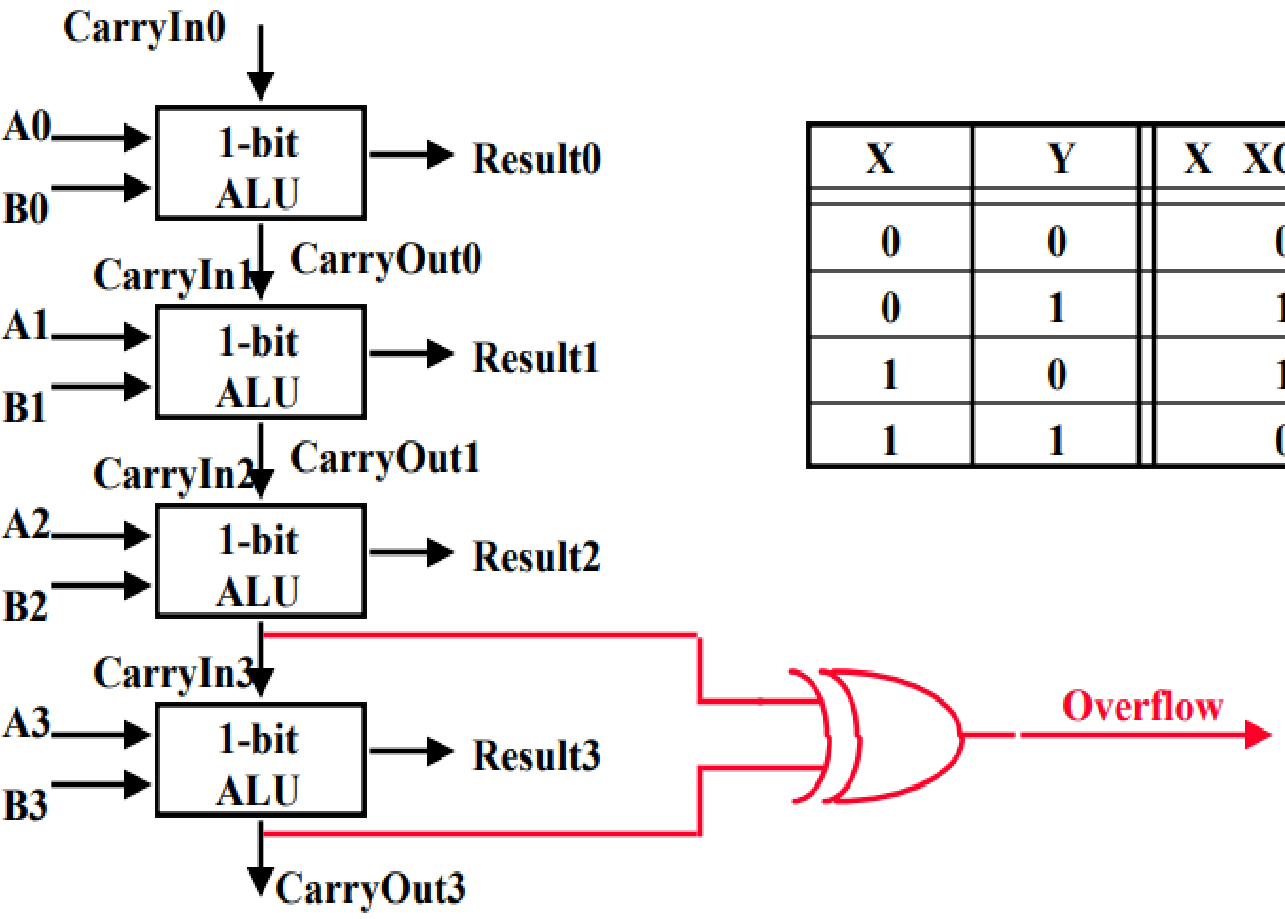
  0 to 2^{n-1} (which is 2^n different values).

- If a signed integer has n bits, it can contain a number between

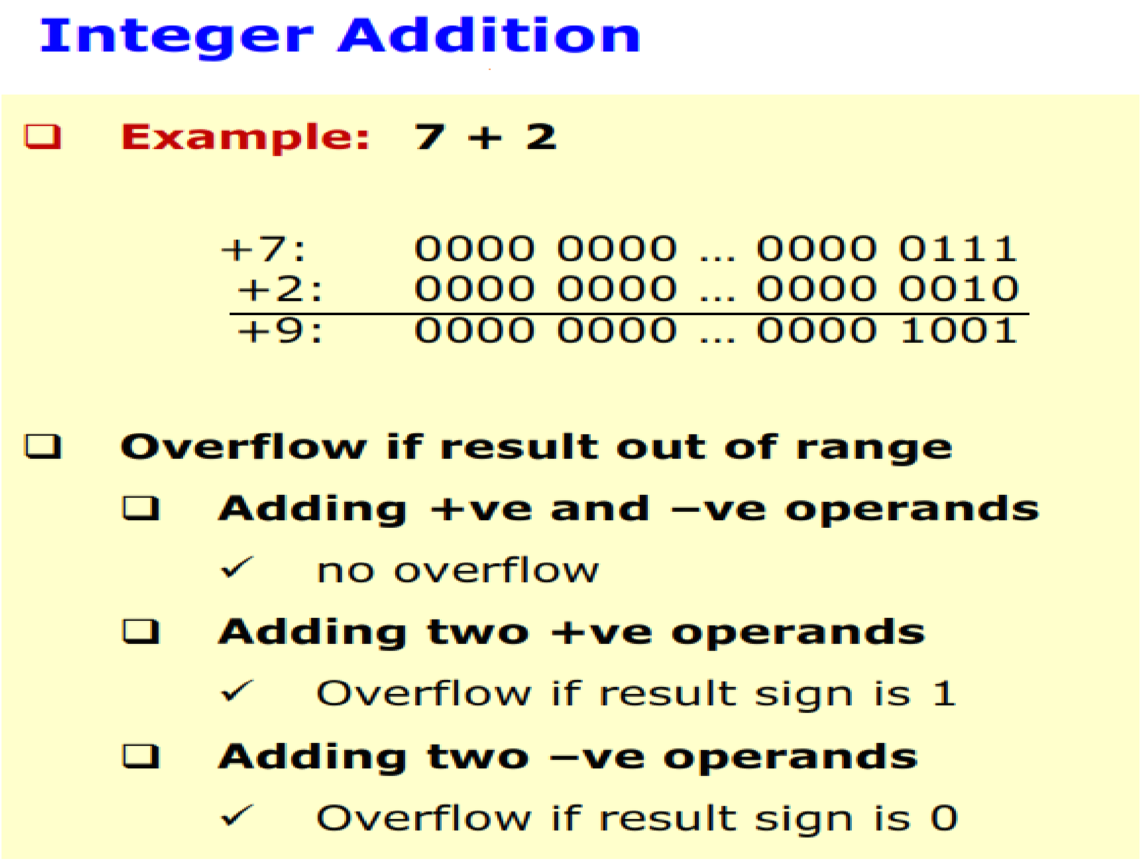
  -2^{n-1} to $+(2^{n-1}-1)$

Overflow Detection Logic

- For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

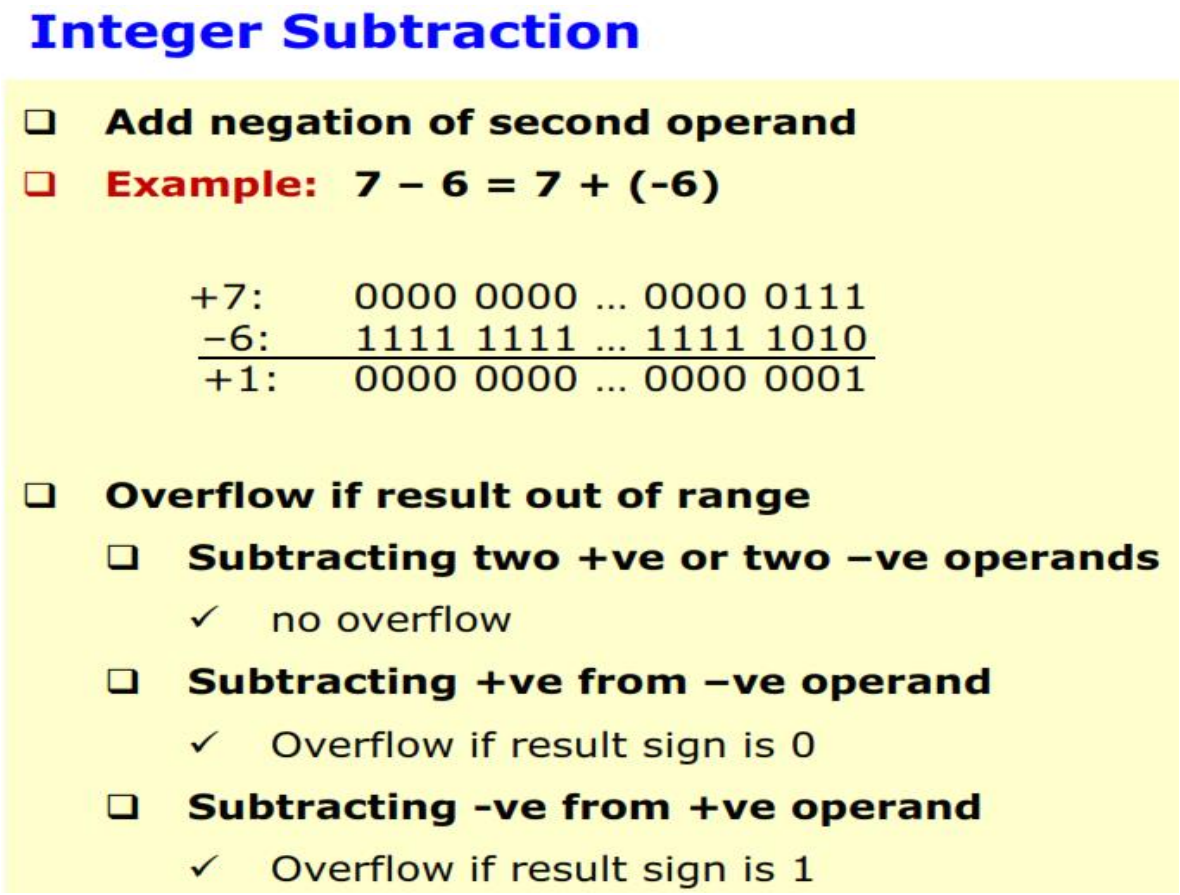


Integer Addition

- □ **Example: 7 + 2**

```
  +7:     0000 0000 ... 0000 0111
   +2:    0000 0000 ... 0000 0010
  ---------------------------------
   +9:    0000 0000 ... 0000 1001
```

- □ **Overflow if result out of range**
  - □ **Adding +ve and −ve operands**
    - ✓ no overflow
  - □ **Adding two +ve operands**
    - ✓ Overflow if result sign is 1
  - □ **Adding two −ve operands**
    - ✓ Overflow if result sign is 0



Integer Subtraction

- □ **Add negation of second operand**
- □ **Example: 7 – 6 = 7 + (-6)**

```
  +7:     0000 0000 ... 0000 0111
   −6:    1111 1111 ... 1111 1010
  ---------------------------------
   +1:    0000 0000 ... 0000 0001
```

- □ **Overflow if result out of range**
  - □ **Subtracting two +ve or two −ve operands**
    - ✓ no overflow
  - □ **Subtracting +ve from −ve operand**
    - ✓ Overflow if result sign is 0
  - □ **Subtracting -ve from +ve operand**
    - ✓ Overflow if result sign is 1

Dealing with Overflow

- The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:
    - Add (**add**), add immediate (**addi**), and subtract (**sub**) cause exceptions (interrupt) on overflow.
    - Add unsigned (**addu**), add immediate unsigned (**addiu**), and subtract unsigned (**subu**) do not cause exceptions (interrupt) on overflow.
- Some languages (e.g., C, Java) ignore overflow
    - The MIPS C compilers use : addu, addui, subu instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
    - The MIPS C Fortran use : add, addi, sub instructions
    - On overflow, invoke exception handler
- **exception** Also called **interrupt** on many computers . An unscheduled event that disrupts program execution; used to detect overflow. Interrupt an exception that comes from outside of the processor.
    - Save PC in exception program counter (EPC) register.
    - Jump to predefined handler address
    - **mfc0** (move from coprocessor reg) instruction is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the off ending instruction via a jump register instruction.

What about Performance?

Full-Adder (FA)

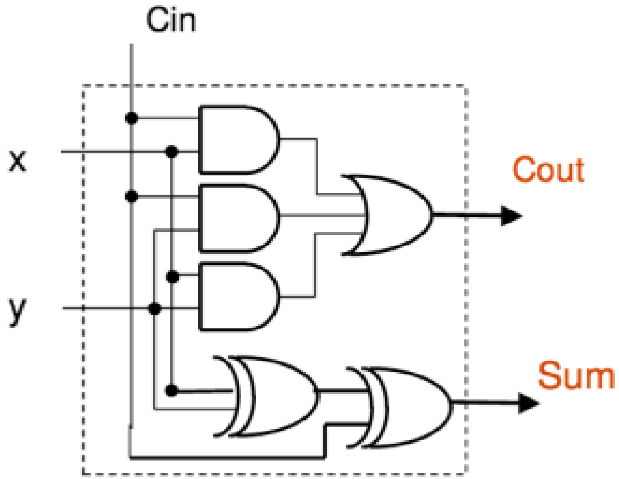
- Examine the Full Adder table

x	y	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Cout = $x \bullet y + Cin \bullet (x + y)$

$S = x'y'c + x'yc' + xy'c' + xyc$

$= x \oplus y \oplus c$

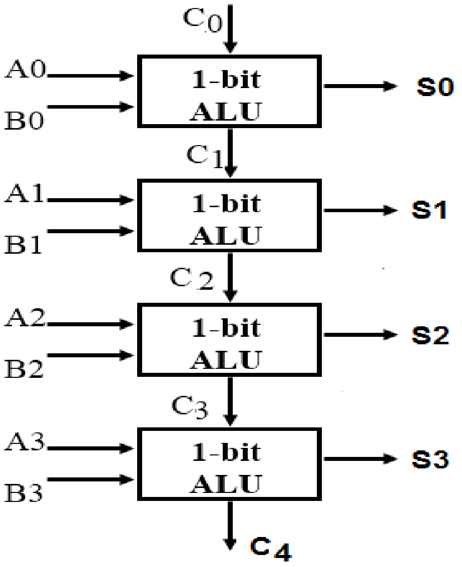


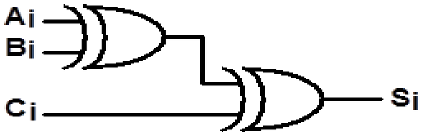
In general, for bit *i*:

$c_{i+1} = x_i y_i + c_i (x_i+y_i)$

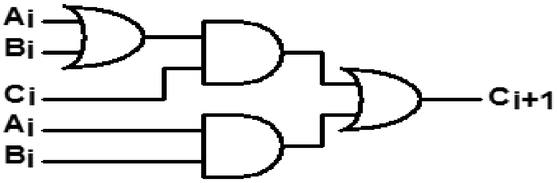
where c_{i+1} = Cout, c_i= Cin

- Critical Path of n-bit Rippled-carry adder is n*CP

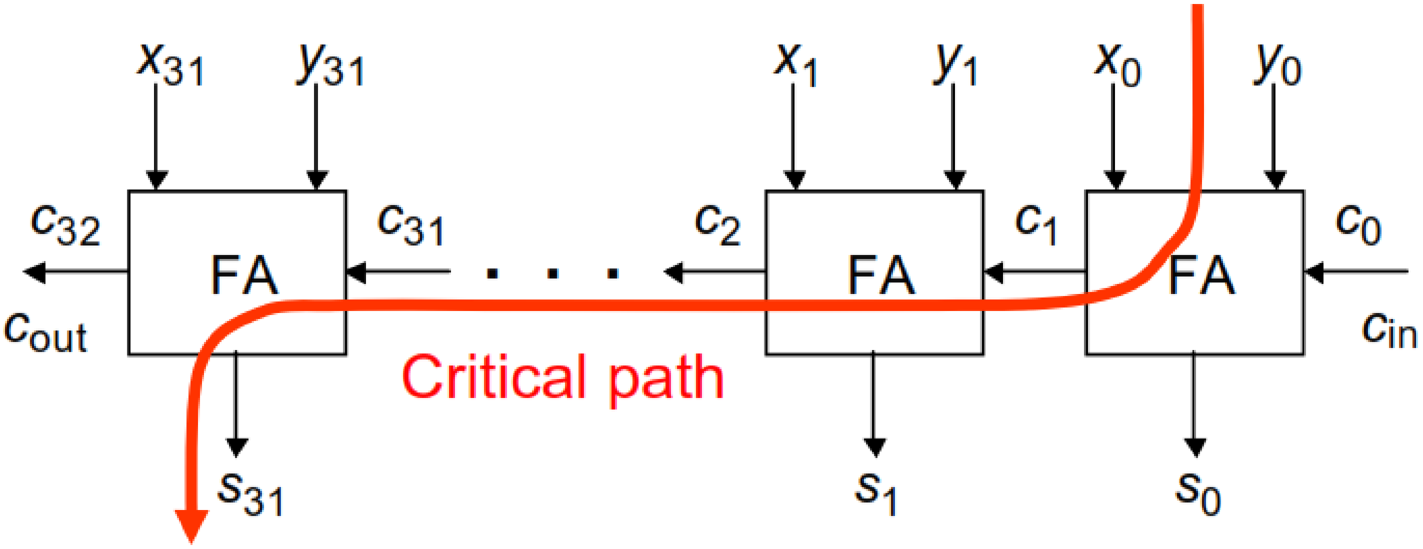




Si= Ai XOR Bi XOR Ci



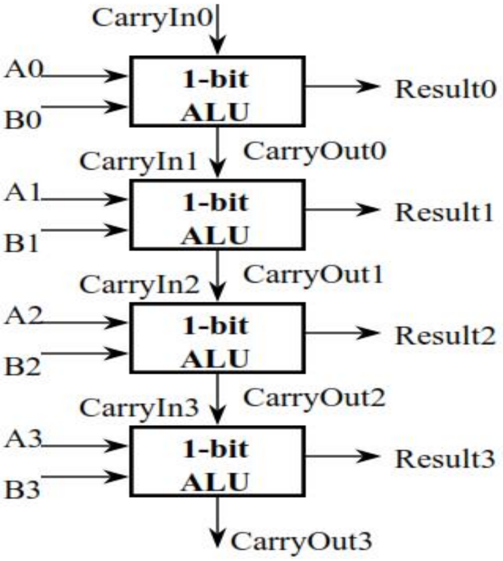
C_{i+1} =($A_i + B_i$)$C_i + A_i . B_i$

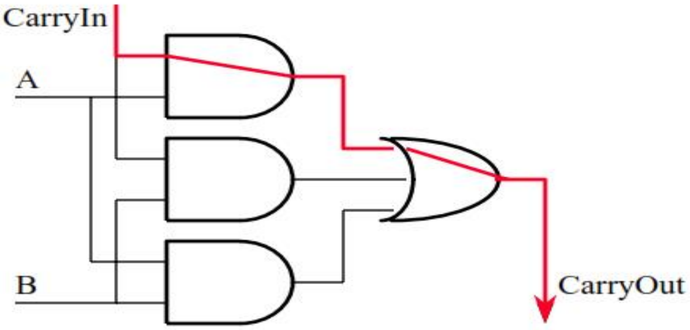


$$\boxed{\begin{aligned} ci + 1 &= (bi \cdot ci) + (ai \cdot ci) + (ai \cdot bi) \\ &= (ai \cdot bi) + (ai + bi) \cdot ci \end{aligned}}$$

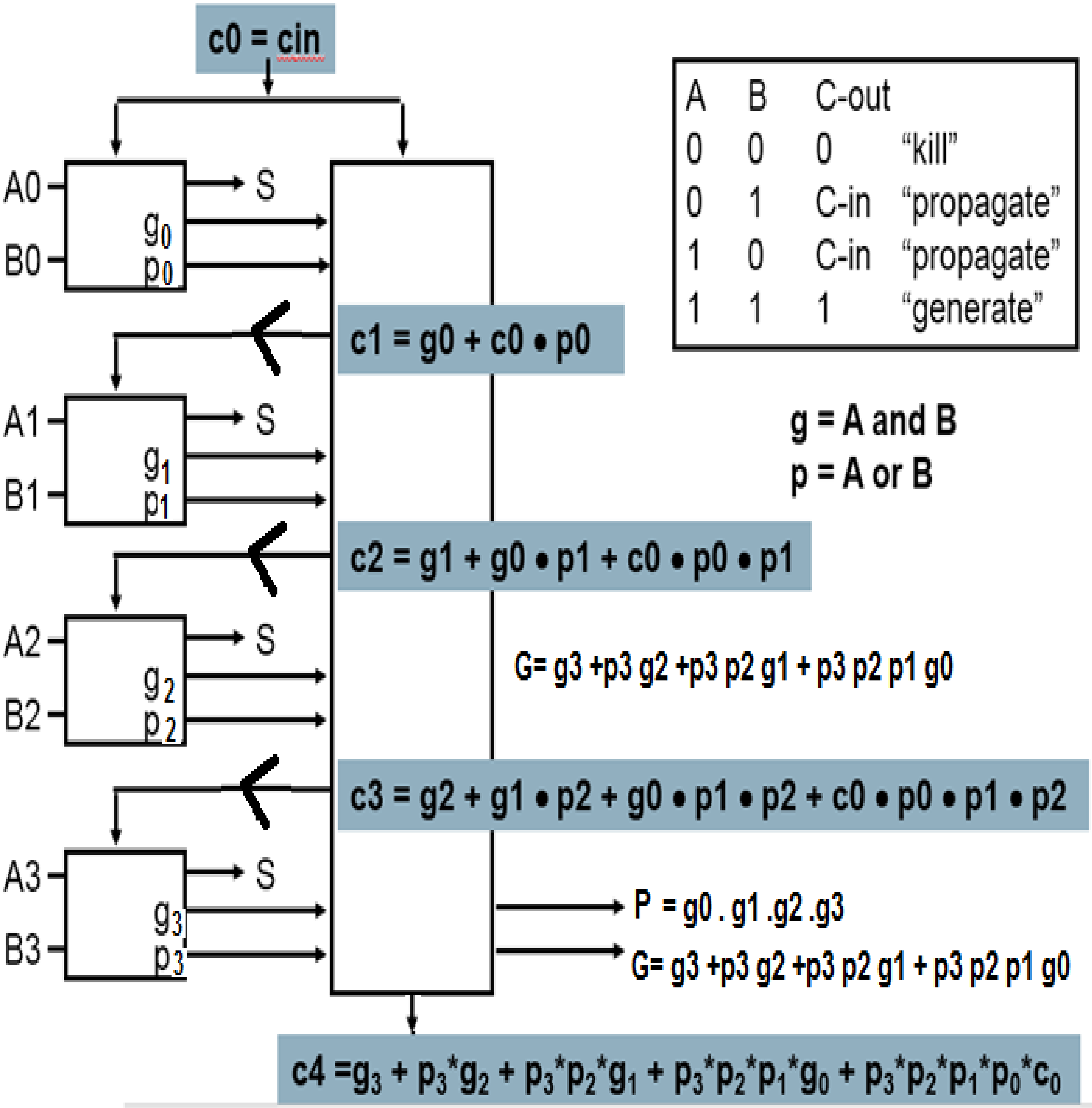
The Disadvantage of Ripple Carry

- The adder we just built is called a “Ripple Carry Adder”
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for a N-bit adder: 2N-gate delay





Carry Look Ahead (Design trick: peek)



c0 = cin

$c1 = g0 + c0 \bullet p0$

$c2 = g1 + g0 \bullet p1 + c0 \bullet p0 \bullet p1$

$c3 = g2 + g1 \bullet p2 + g0 \bullet p1 \bullet p2 + c0 \bullet p0 \bullet p1 \bullet p2$

A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

g = A and B
p = A or B

$G= g3 +p3\ g2 +p3\ p2\ g1 + p3\ p2\ p1\ g0$

$P = g0 . g1 .g2 .g3$

$G= g3 +p3\ g2 +p3\ p2\ g1 + p3\ p2\ p1\ g0$

$c4 =g_3 + p_3{}^*g_2 + p_3{}^*p_2{}^*g_1 + p_3{}^*p_2{}^*p_1{}^*g_0 + p_3{}^*p_2{}^*p_1{}^*p_0{}^*c_0$

$$gi = ai \cdot bi$$
$$pi = ai + bi$$

Using them to define $ci + 1$, we get

$$ci + 1 = gi + pi \cdot ci$$

To see where the signals get their names, suppose gi is 1. Then

$$ci + 1 = gi + pi \cdot ci = 1 + pi \cdot ci = 1$$

That is, the adder *generates* a CarryOut ($ci + 1$) independent of the value of CarryIn (ci). Now suppose that gi is 0 and pi is 1. Then

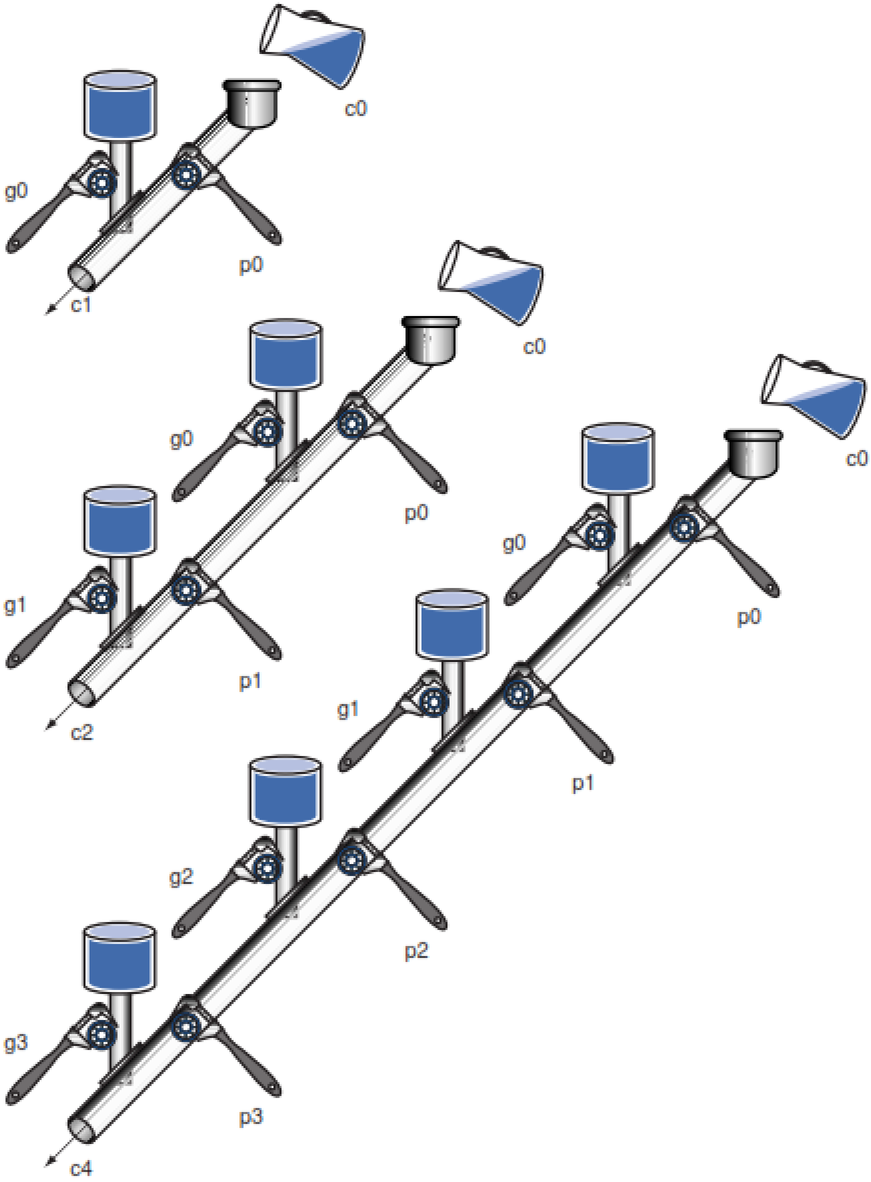
$$ci + 1 = gi + pi \cdot ci = 0 + 1 \cdot ci = ci$$

That is, the adder *propagate*s CarryIn to a CarryOut. Putting the two together, CarryIni + *1* is a 1 if either gi is 1 or both pi is 1 and CarryIni is 1.

$$\begin{aligned}
c1 &= g0 + (p0 \cdot c0) \\
c2 &= g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0) \\
c3 &= g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0) \\
c4 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\
&\quad + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)
\end{aligned}$$

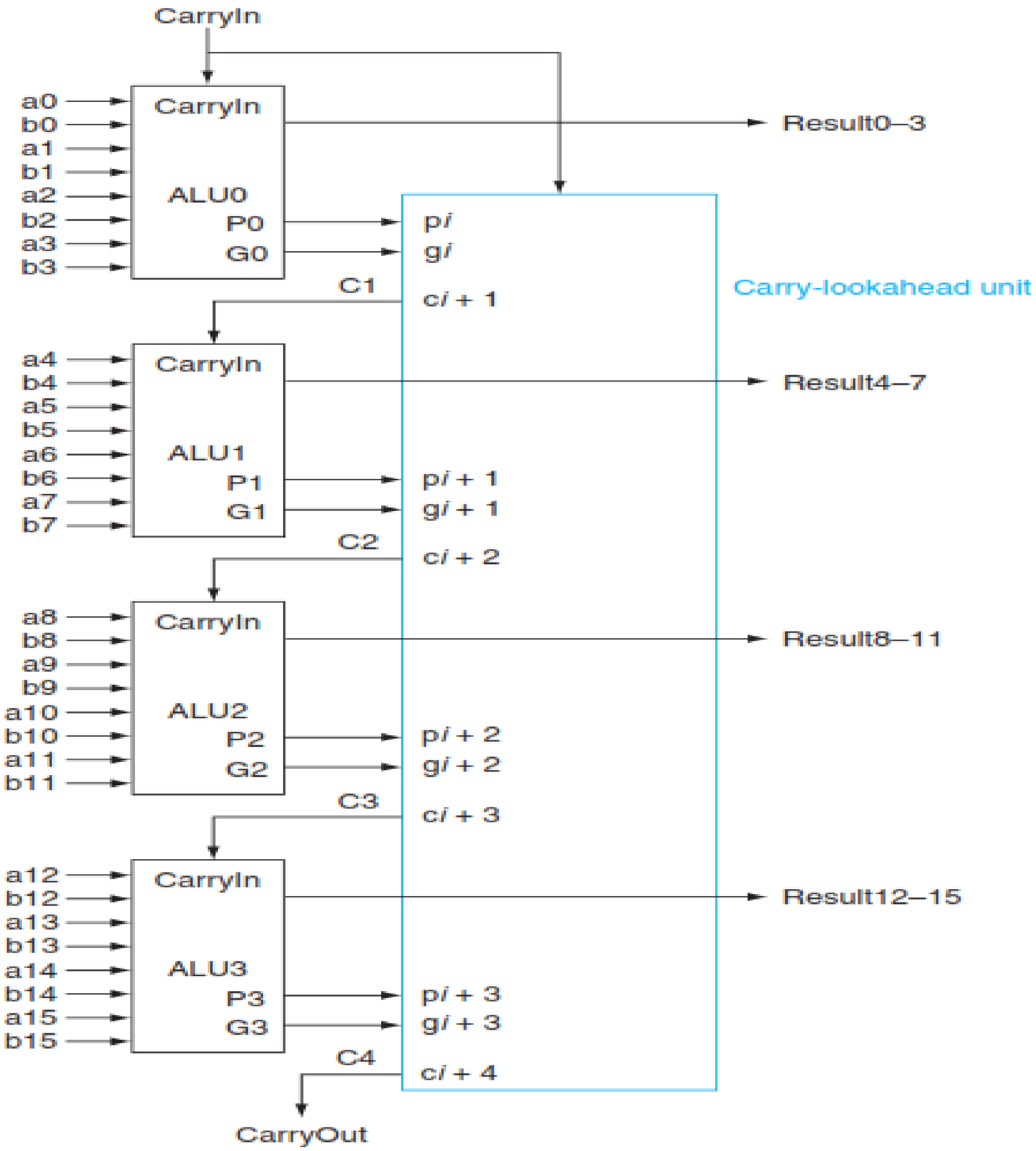
A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water pipes and valves.

$$\begin{aligned}
c1 &= g0 + (p0 \cdot c0) \\
c2 &= g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0) \\
c3 &= g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0) \\
c4 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\
&\quad + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)
\end{aligned}$$



Group Carry Look-ahead (16-bit): Abstraction

Four 4-bit ALUs using carry lookahead to form a 16-bit adder. Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.



For the first 4 bit ALU (ALU0))

P0= p0 .p1 .p2 .p3

G0= $g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0$

Cout = G0+P0.CarryIn

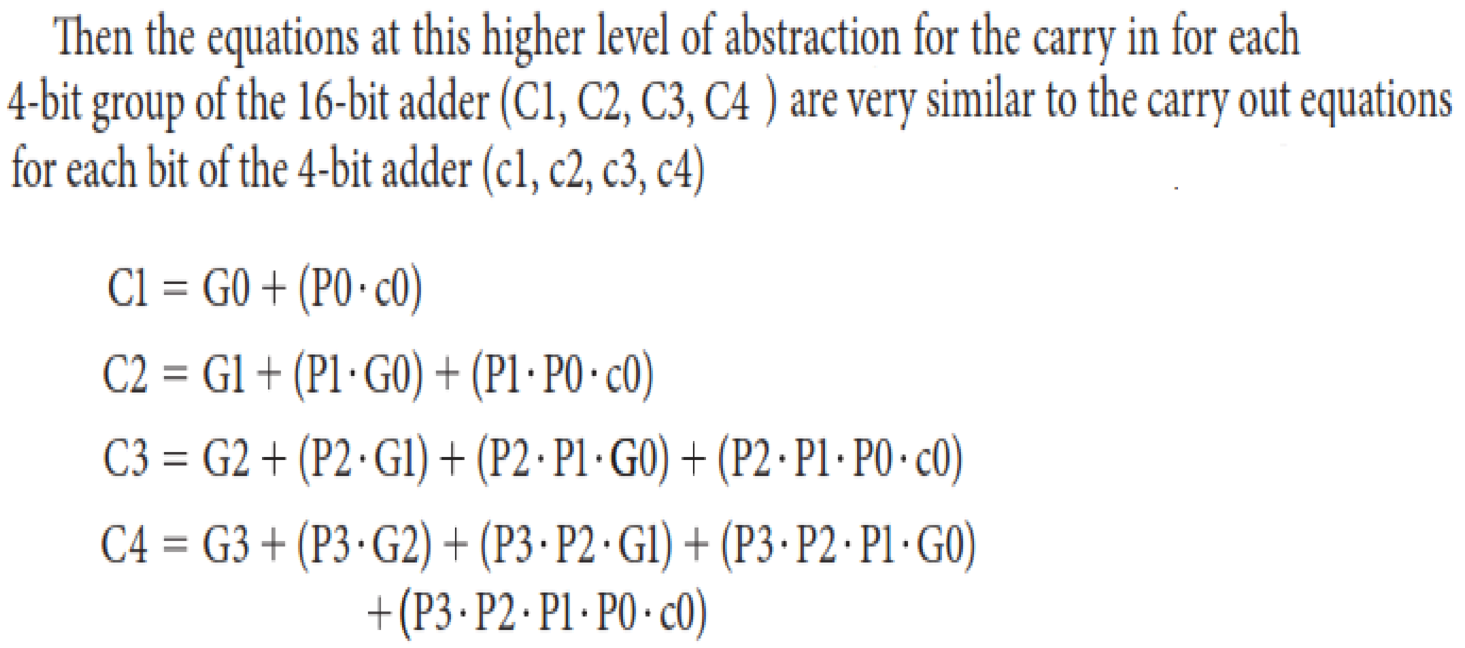
Group Carry-Lookahead

- $c_4 = g_3 + p_3*g_2 + p_3*p_2*g_1 + p_3*p_2*p_1*g_0 + p_3*p_2*p_1*p_0*c_0$
- Approach: use carry lookahead for 4-bit groups
  - "Super Propagate" equations:

$$P_0 = p_3*p_2*p_1*p_0$$
$$P_1 = p_7*p_6*p_5*p_4$$
$$P_2 = p_{11}*p_{10}*p_9*p_8$$
$$P_3 = p_{15}*p_{14}*p_{13}*p_{12}$$

  - "Super Generate" equations:

$$G_0 = g_3 + (p_3*g_2) + (p_3*p_2* g_1) + (p_3*p_2 *p_1 * g_0)$$
$$G_1 = g_7 + (p_7*g_6) + (p_7*p_6* g_5) + (p_7*p_6 *p_5 * g_4)$$
$$G_2 = g_{11} + (p_{11}*g_{10}) + (p_{11}*p_{10}* g_9) + (p_{11}*p_{10} *p_9 * g_8)$$
$$G_3 = g_{15} + (p_{15}*g_{14}) + (p_{15}*p_{14}* g_{13}) + (p_{15}*p_{14} *p_{13} * g_{12})$$

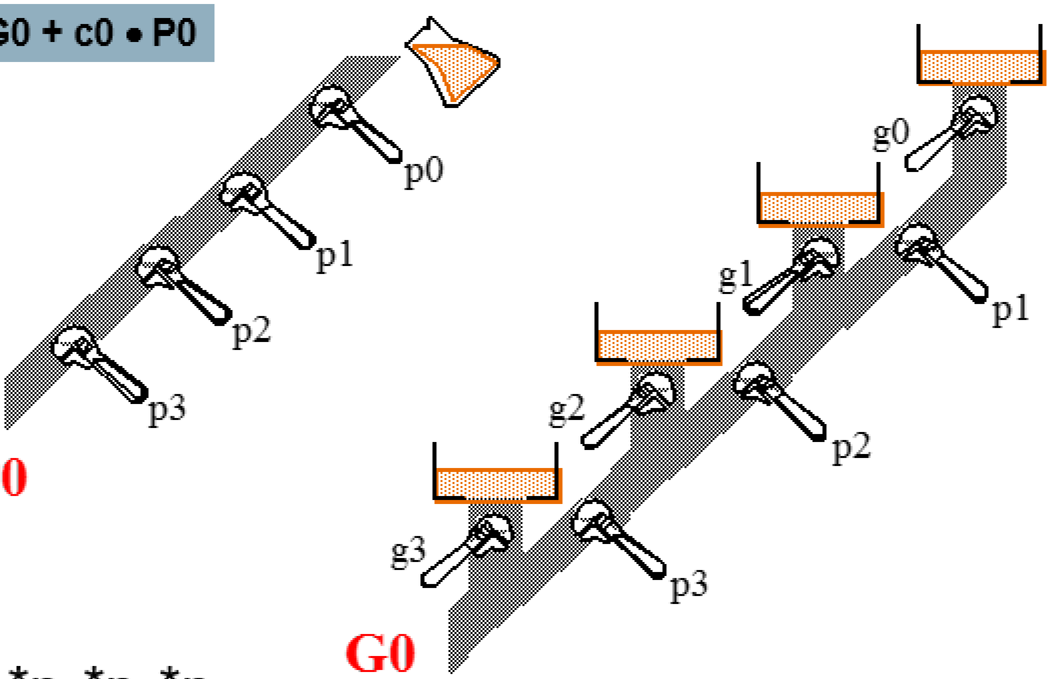


Then the equations at this higher level of abstraction for the carry in for each 4-bit group of the 16-bit adder (C1, C2, C3, C4) are very similar to the carry out equations for each bit of the 4-bit adder (c1, c2, c3, c4)

$$C1 = G0 + (P0 \cdot c0)$$
$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$
$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$
$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

2nd level Carry, Propagate as Plumbing

C1 = G0 + c0 • P0

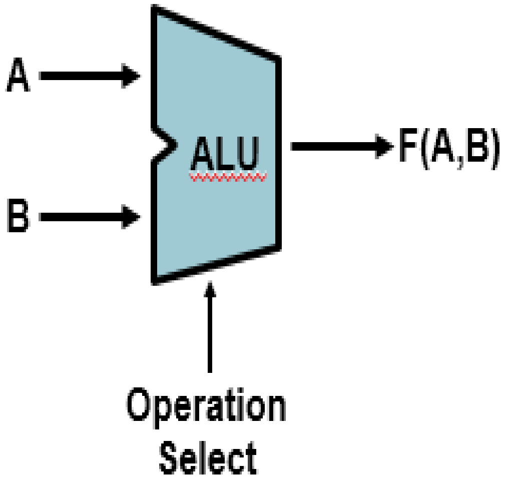


$$P_0 = p_3{}^*p_2{}^*p_1{}^*p_0$$

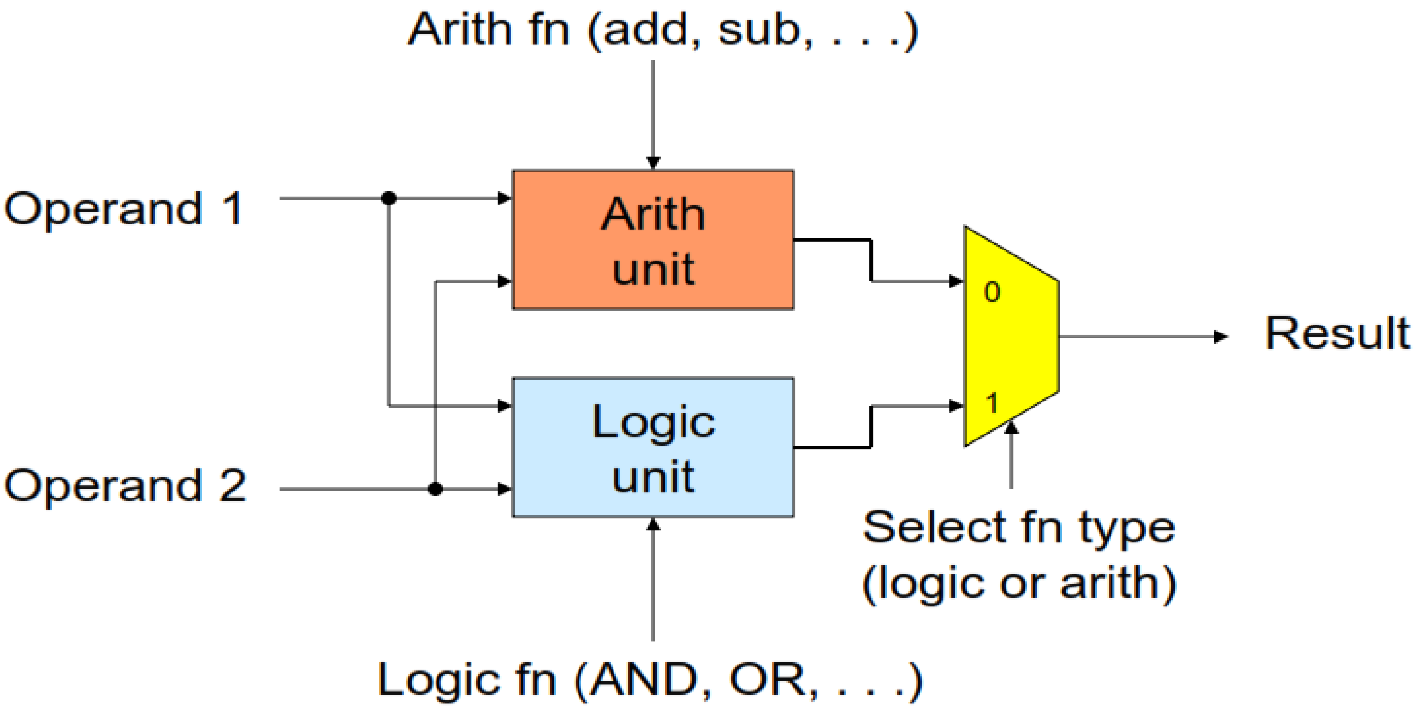
$$G_0 = g_3 + (p_3{}^*g_2) + (p_3{}^*p_2{}^*\, g_1) + (p_3{}^*p_2{}^*p_1{}^*\, g_0)$$

Arithmetic-Logic Units

- Combinational logic element that performs multiple functions:
  - Arithmetic: add, subtract
  - Logical: AND, OR, & NOR
- Gates, multiplexer for logic
- functions & adder



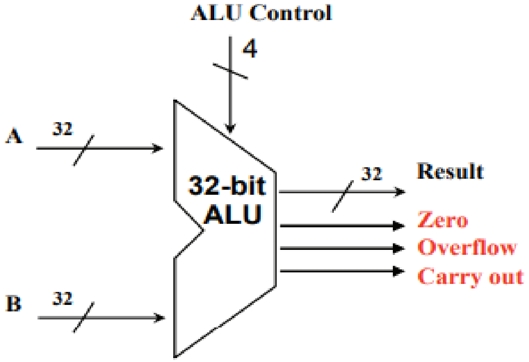
Multifunction ALUs


Arith fn (add, sub, . . .)
Operand 1
Arith unit
0
Result
Logic unit
1
Operand 2
Select fn type (logic or arith)
Logic fn (AND, OR, . . .)

General structure of a simple arithmetic/logic unit.

Functioning of 32-bit ALU

	ALU Control lines		
Function	**Ainvert**	**Binvert**	**Operation**
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
slt	0	1	11
nor	1	1	00

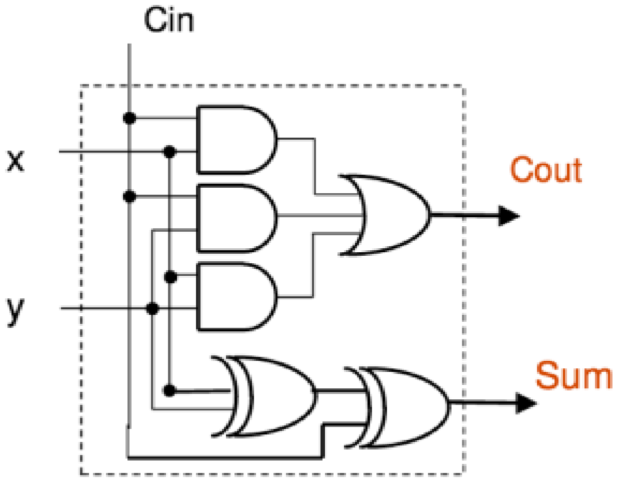


- Result lines provide result of the chosen function applied to values of A and B
- Since this ALU operates on 32-bit operands, it is called 32-bit ALU
- Zero output indicates if all Result lines have value 0
- Overflow indicates integer overflow of add and subtract functions; for unsigned integers, this overflow indicator does not provide any useful information
- Carry out indicates carry out and unsigned integer overflow

Full-Adder (FA)

- Examine the Full Adder table

x	y	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



In general, for bit i:

$$c_{i+1} = x_i\, y_i + c_i\,(x_i+y_i)$$

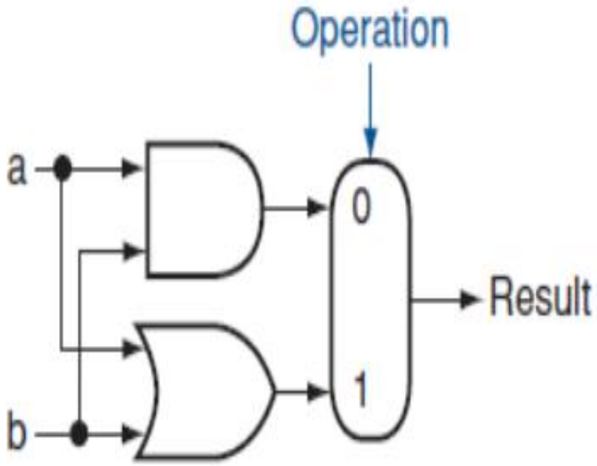
where c_{i+1} = Cout, c_i= Cin

$$Cout = x \bullet y + Cin \bullet (x + y)$$

$$S = x'y'c + x'yc' + xy'c' + xyc$$
$$= x \oplus y \oplus c$$

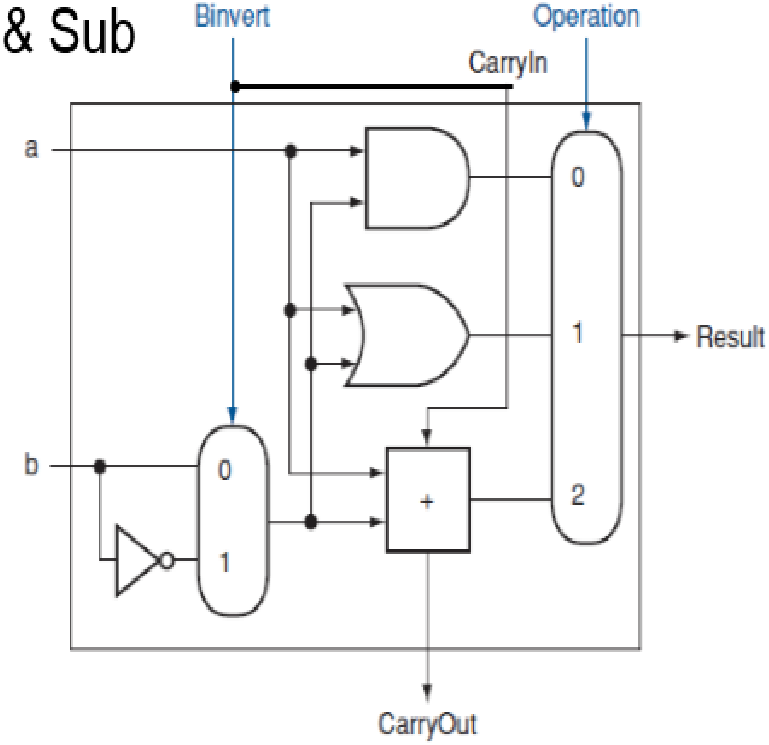
A 1-Bit ALU

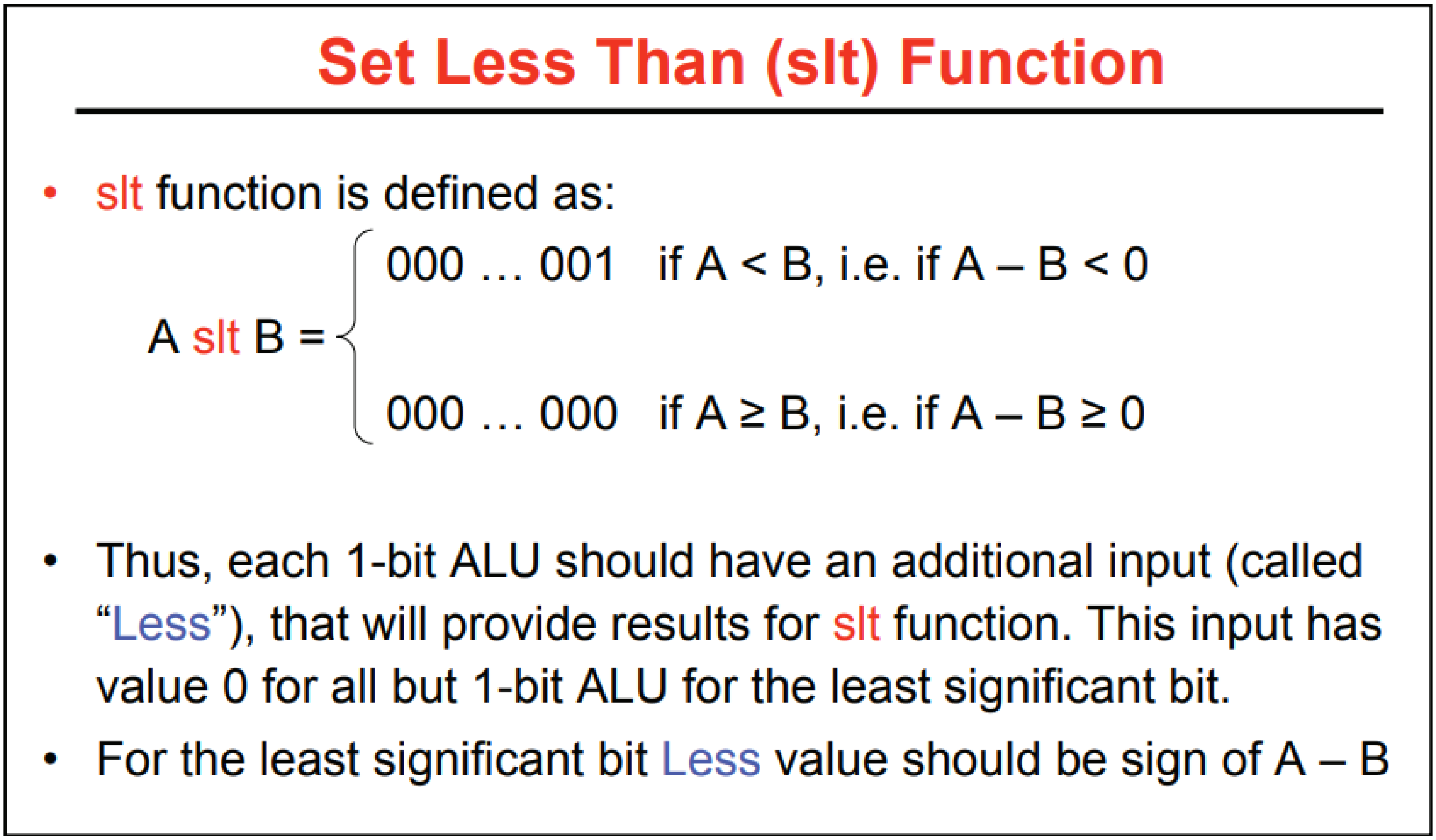
- The 1-bit logical unit for AND and OR.



A 1-Bit ALU (Subtraction)

- A 1-bit ALU that performs AND, OR, Add & Sub





Set Less Than (slt) Function

- slt function is defined as:

$$A \text{ slt } B = \begin{cases} 000 \ldots 001 & \text{if } A < B \text{, i.e. if } A - B < 0 \\ 000 \ldots 000 & \text{if } A \geq B \text{, i.e. if } A - B \geq 0 \end{cases}$$

- Thus, each 1-bit ALU should have an additional input (called “Less”), that will provide results for slt function. This input has value 0 for all but 1-bit ALU for the least significant bit.
- For the least significant bit Less value should be sign of A – B

```
SLT  $R, $A, $B
    if ($A < $B)
            $R = 32'b0.....01;
    else
            $R = 32'b0.....00;
```

$\underbrace{0 \cdots 0}$ ↳ upper 31 bits are 0

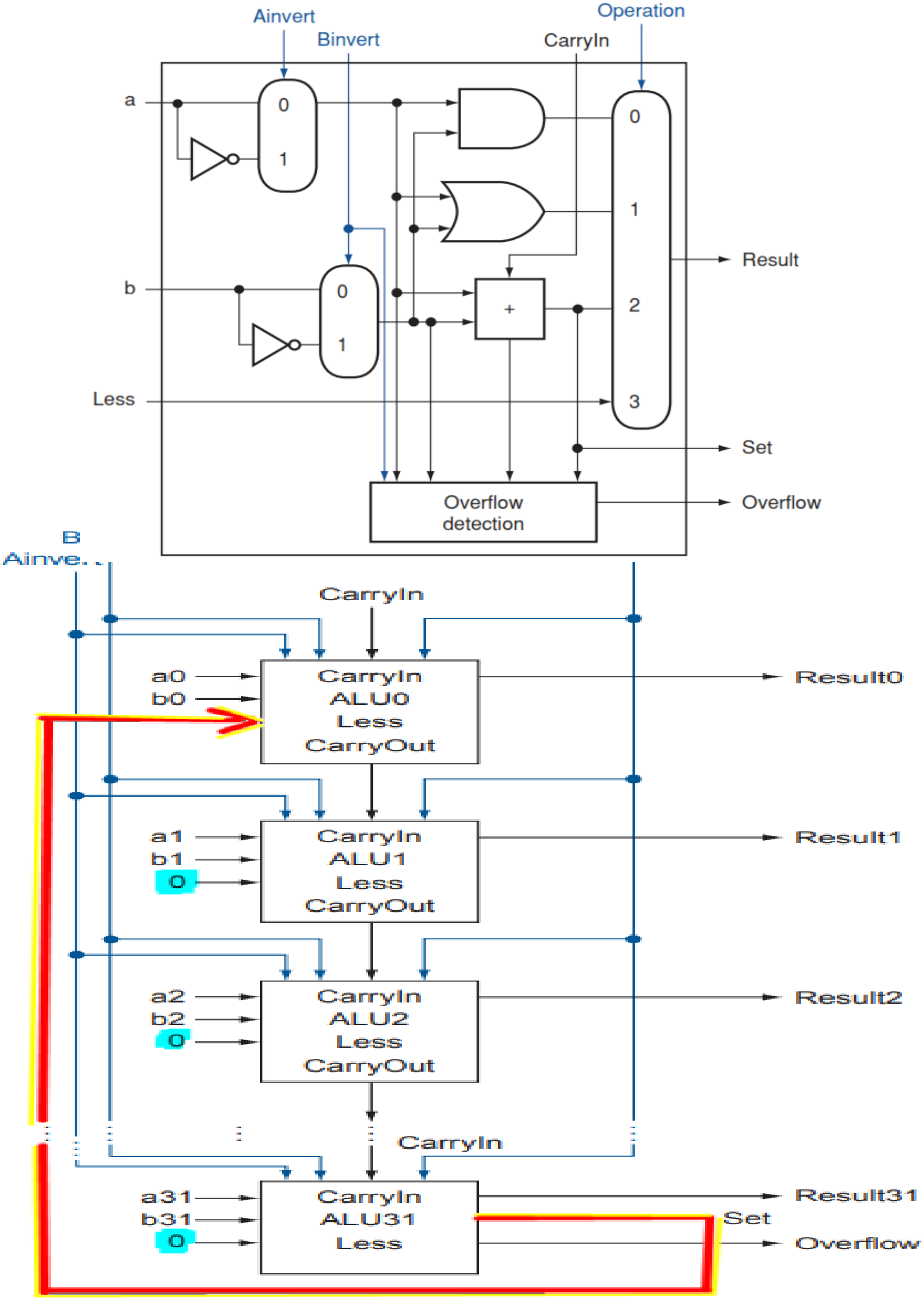
SLT is implemented using SUBTRACTION.

if ($A − $B) is negative

⟹ $A < $B .

$d_{31} = 1$ ⟹ difference is negative

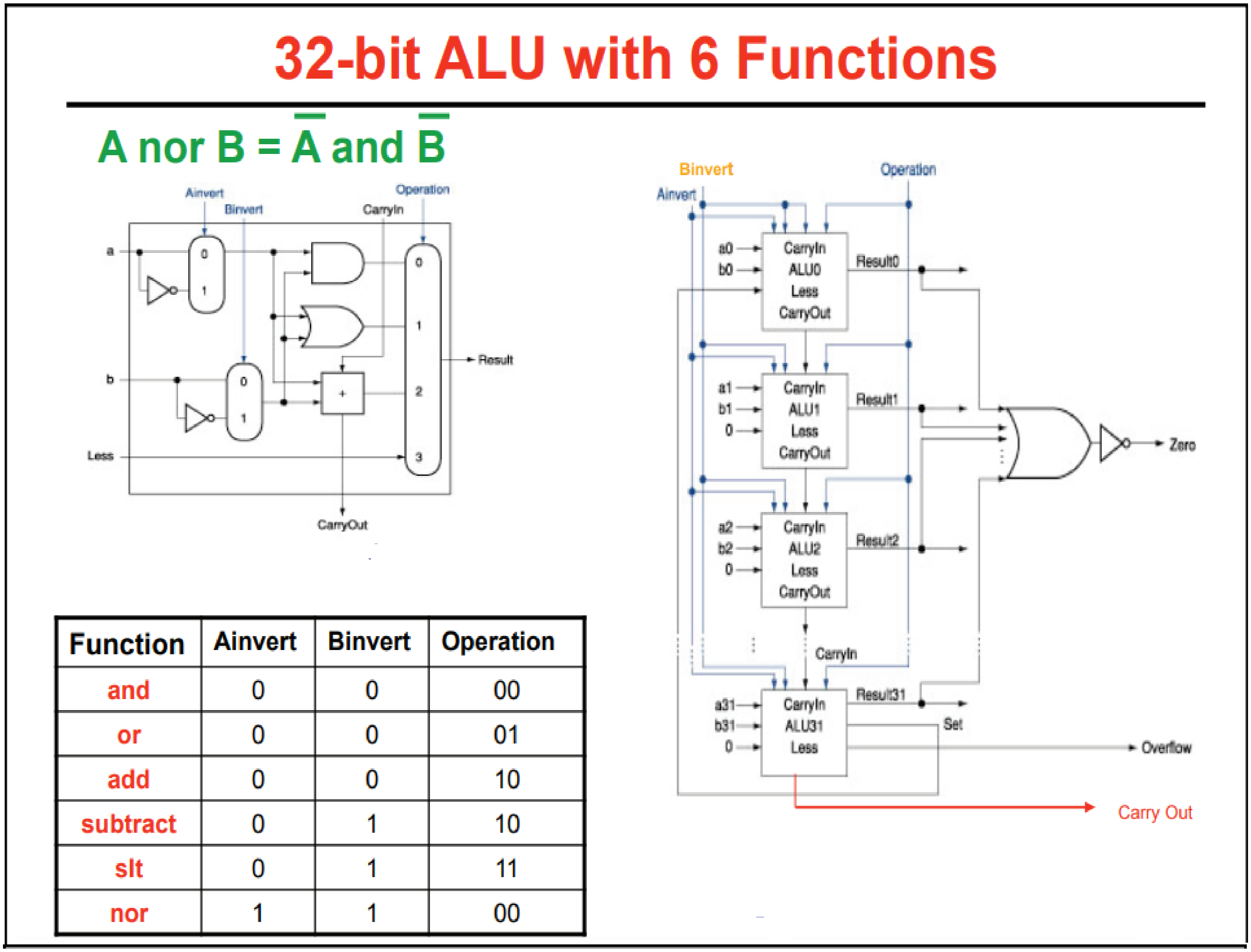
,
1-bit ALU for the most significant bit. The direct output from the (last) adder for the less than comparison called Set.



Support conditional branch instructions

$$(a - b = 0) \Rightarrow a = b$$

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0})}$$

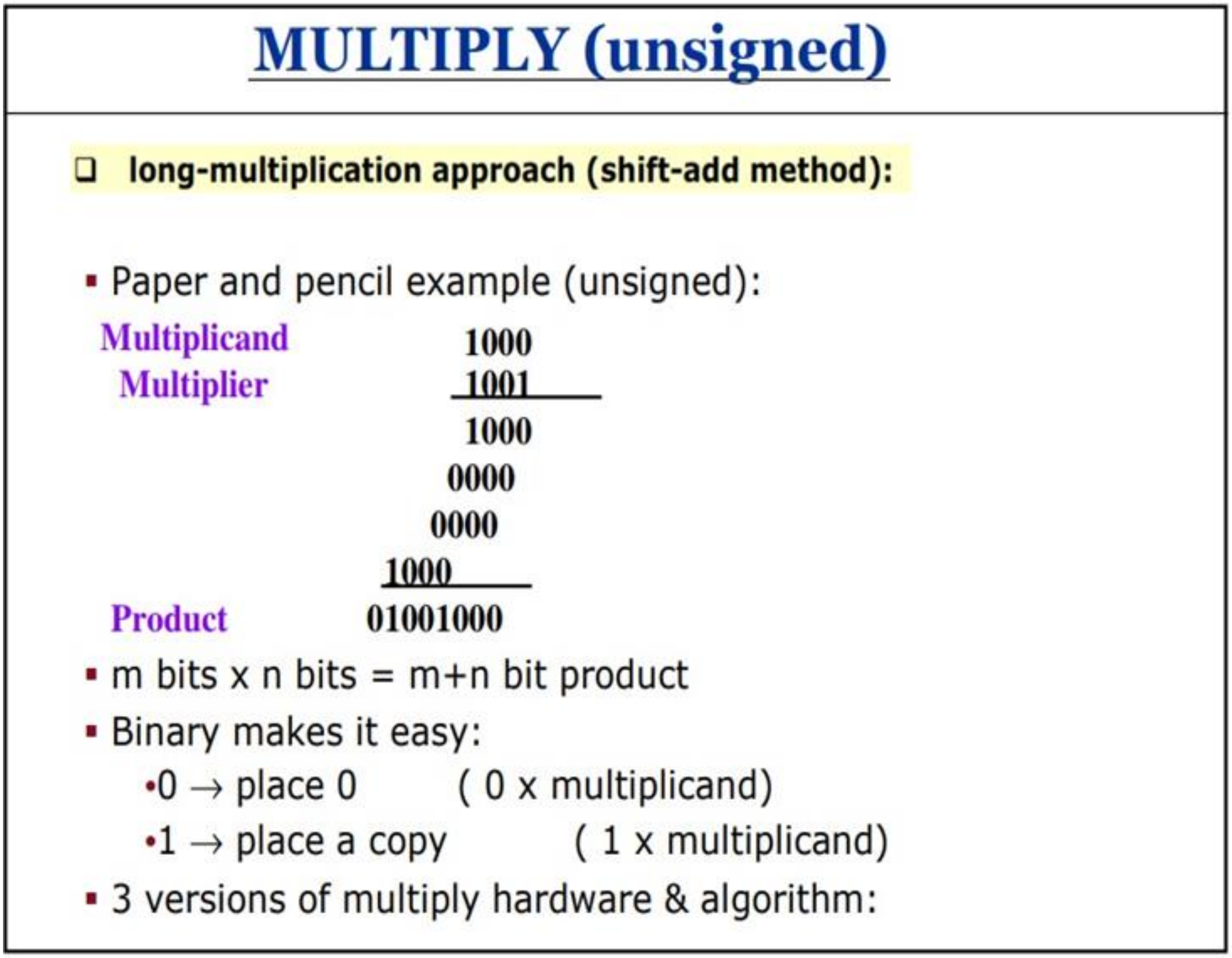


32-bit ALU with 6 Functions

A nor B = $\overline{A}$ and $\overline{B}$

Function	Ainvert	Binvert	Operation
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
slt	0	1	11
nor	1	1	00

Multiplication



MULTIPLY (unsigned)

❑ **long-multiplication approach (shift-add method):**

- Paper and pencil example (unsigned):

```
Multiplicand        1000
 Multiplier         1001
                    ----
                    1000
                   0000
                  0000
                 1000
                 -------
Product          01001000
```

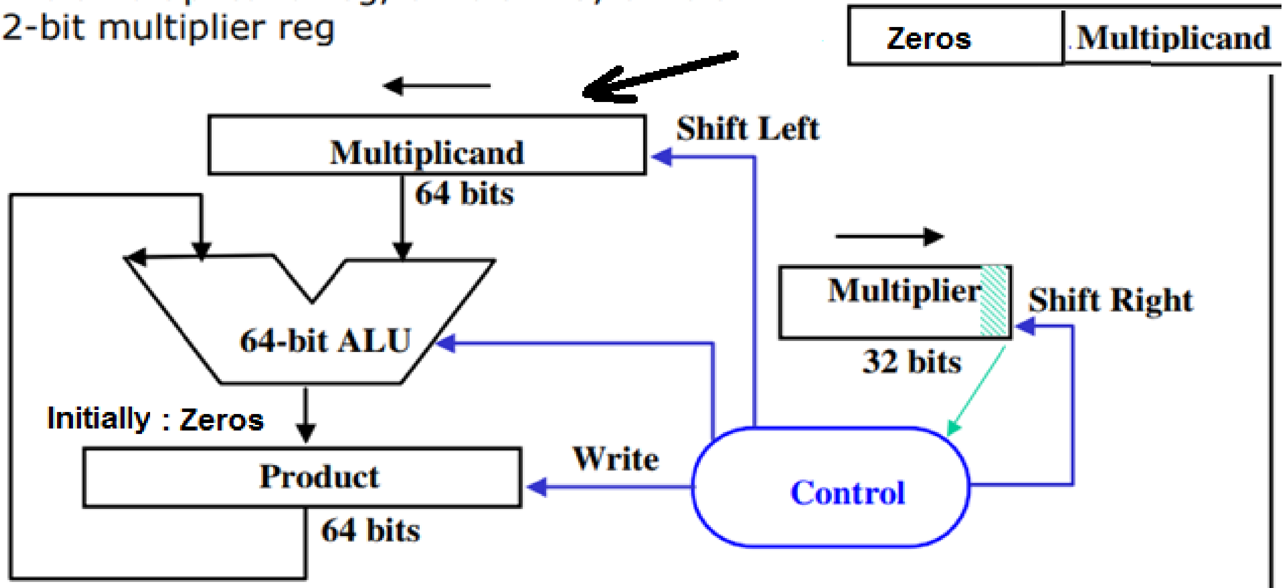
- m bits x n bits = m+n bit product
- Binary makes it easy:
  - •0 → place 0 (0 x multiplicand)
  - •1 → place a copy (1 x multiplicand)
- 3 versions of multiply hardware & algorithm:

❖ **Accomplished via shifting and addition**

❖ **Consumes more time and more chip area than addition**

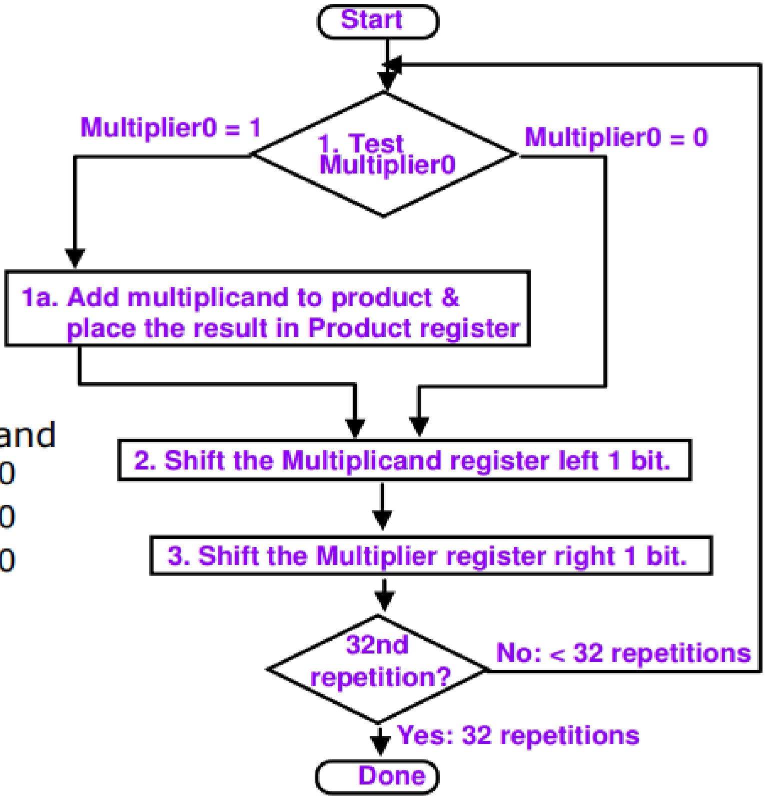
Unsigned shift-add multiplier (version 1)

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg. 32-bit multiplier reg

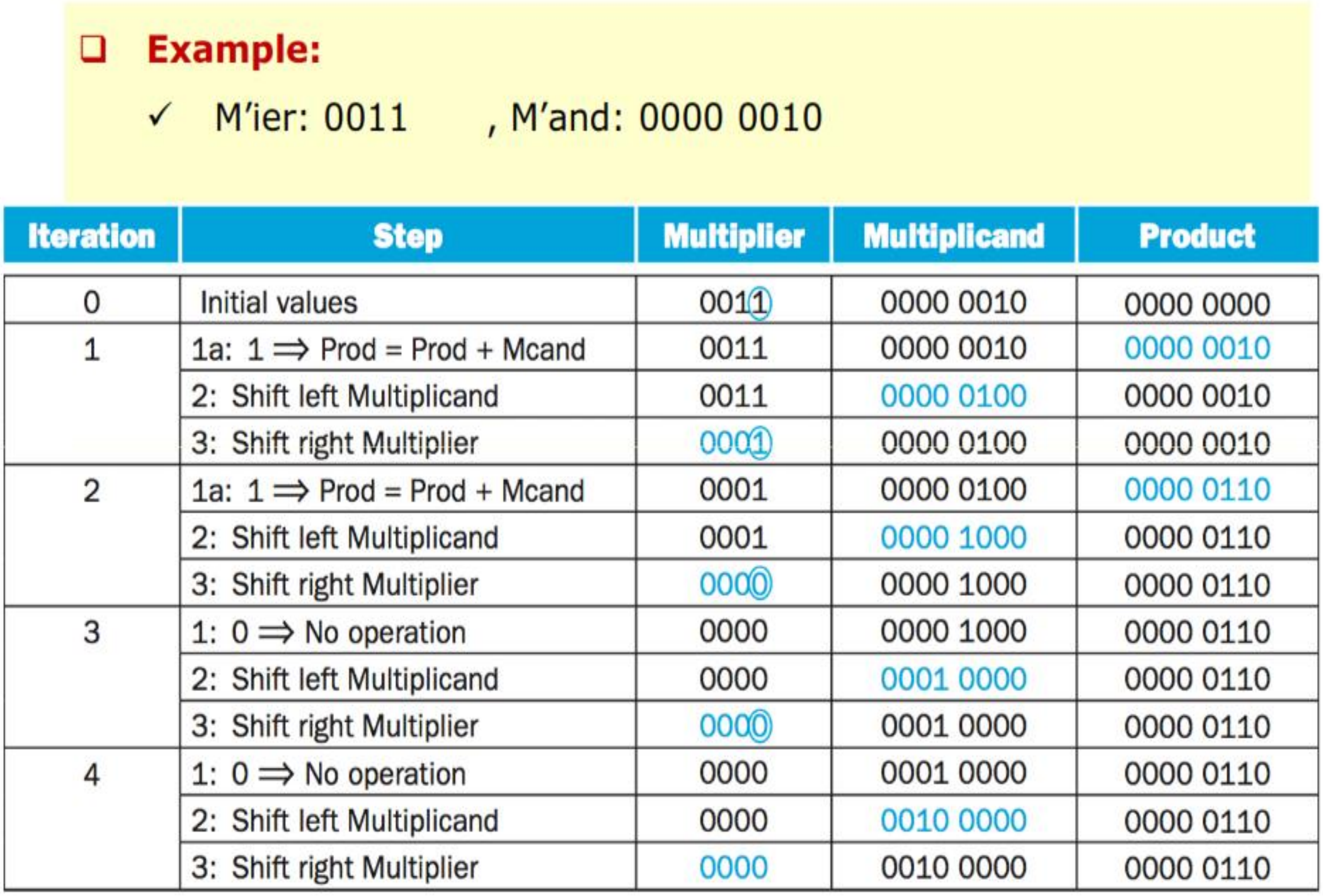


Multiplier = datapath + control

Multiply Algorithm - Version 1


Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product & place the result in Product register
2. Shift the Multiplicand register left 1 bit.
3. Shift the Multiplier register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

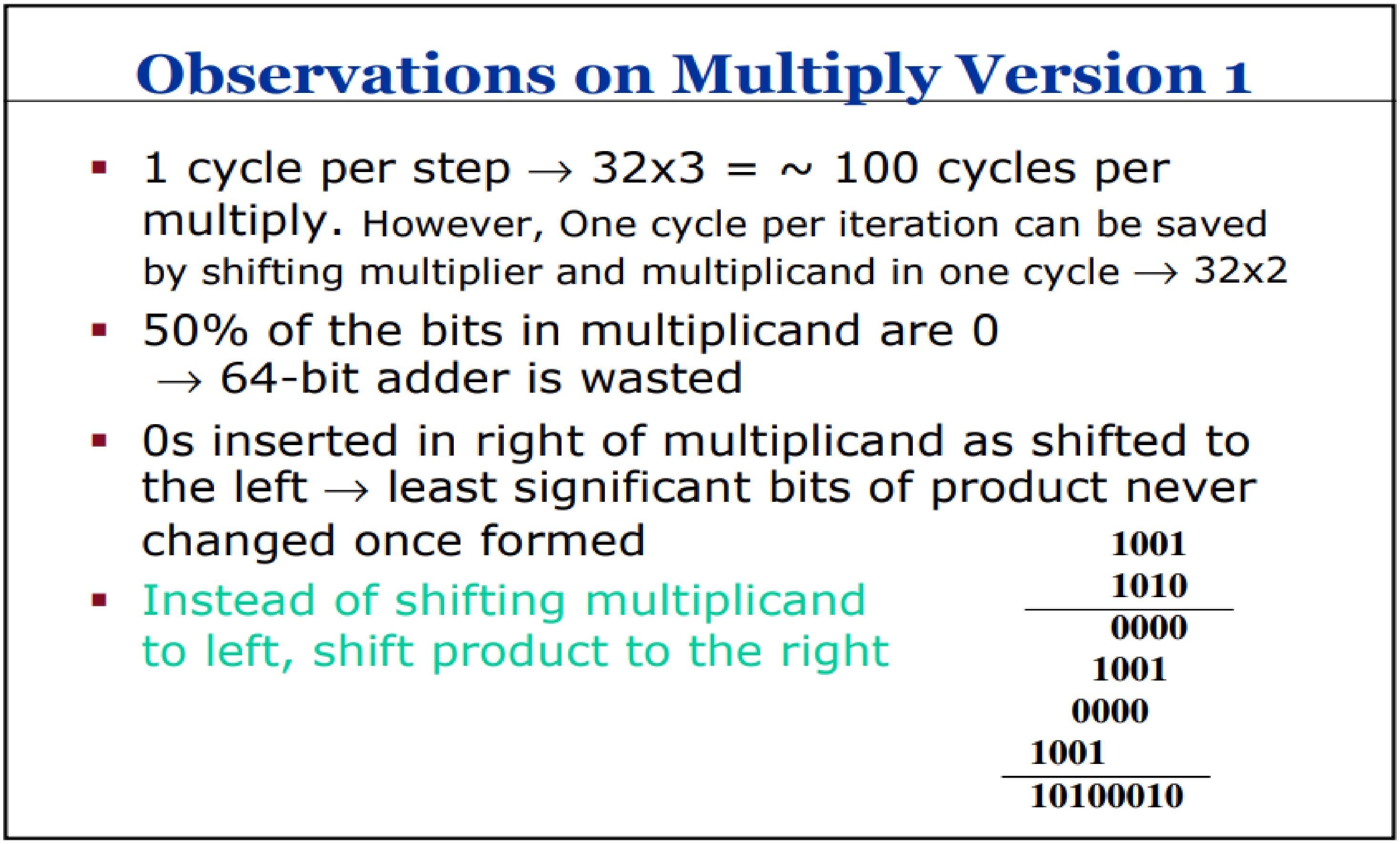
Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110		



❑ **Example:**

✓ M’ier: 0011 , M’and: 0000 0010

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110



Observations on Multiply Version 1

- 1 cycle per step → 32x3 = ~ 100 cycles per multiply. However, One cycle per iteration can be saved by shifting multiplier and multiplicand in one cycle → 32x2
- 50% of the bits in multiplicand are 0 → 64-bit adder is wasted
- 0s inserted in right of multiplicand as shifted to the left → least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to the right

```
    1001
    1010
  --------
    0000
   1001
  0000
 1001
 --------
 10100010
```

Example 6-bit x 6-bit 1st Version multiplier (58 × 23)

Multiplicand = 58 = unsigned 6-bit = $(111010)_2$

Multiplier = 23 = unsigned 6-bit = $(010111)_2$

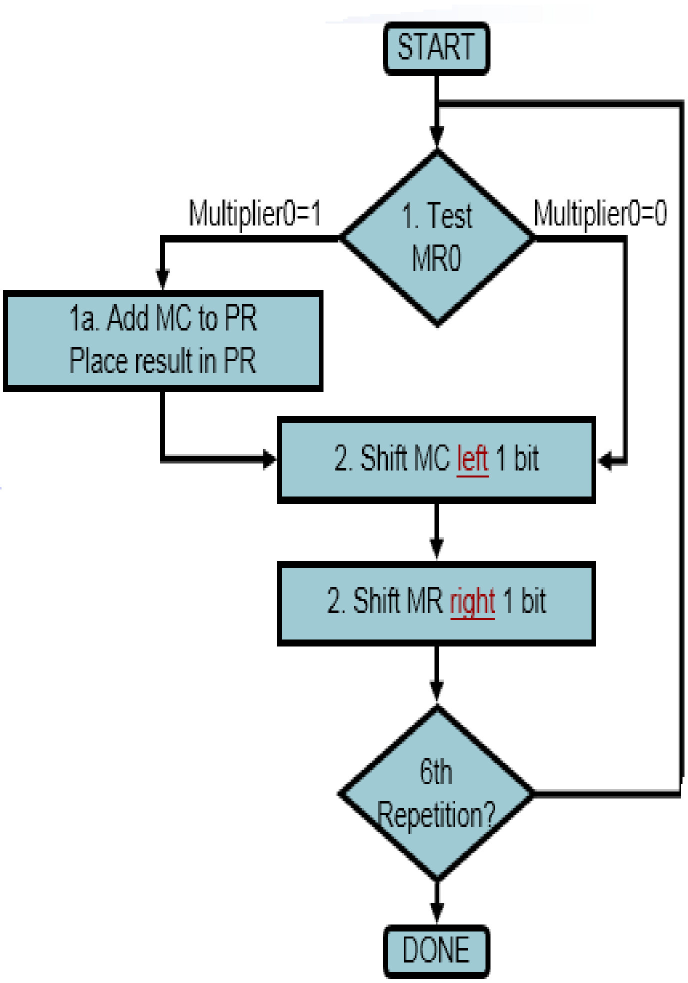
Product = 58 × 23 = 1334 = $(010100110110)_2$

Initial Values:

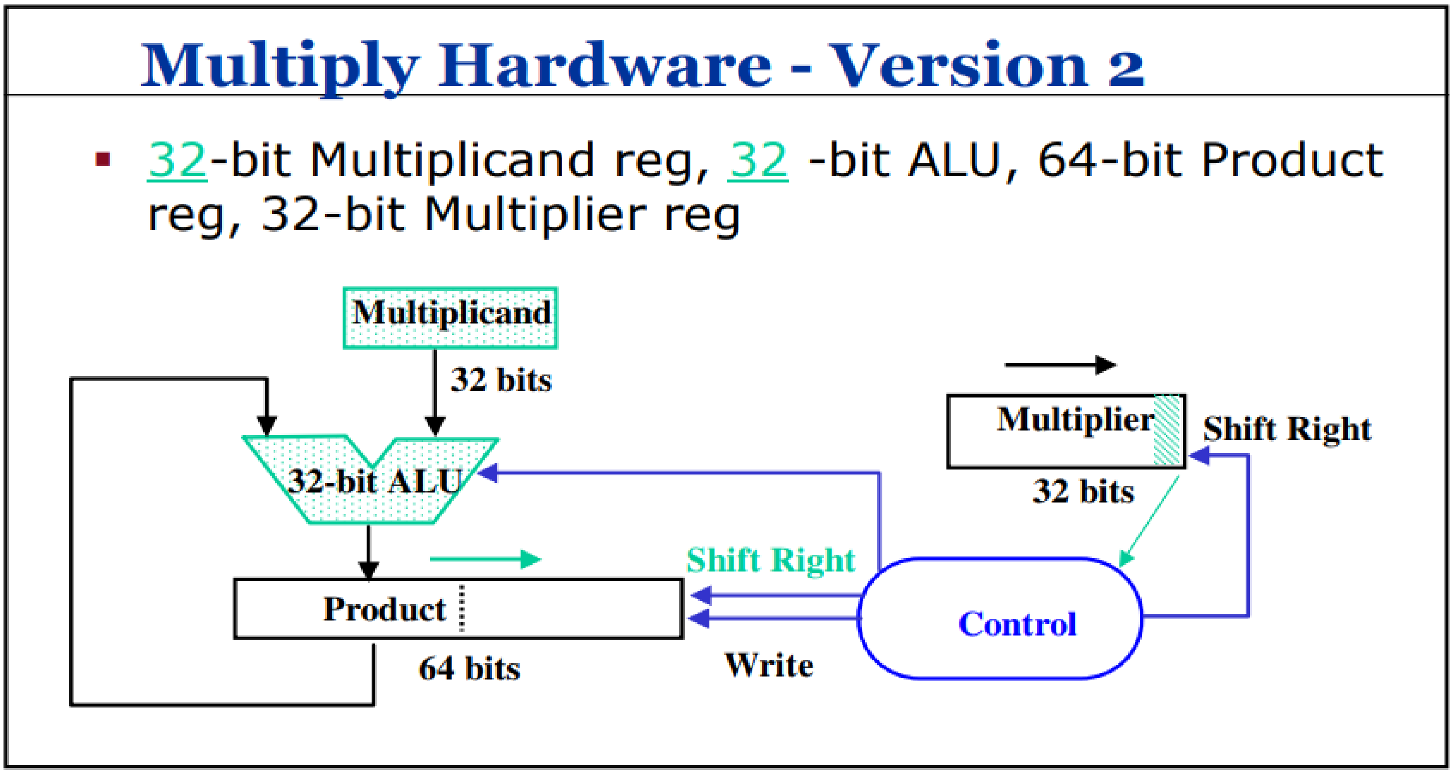
Multiplicand Register = MC is 12 bits = 000000111010.

Multiplier Register = MR is 6 bits = 010111.

Product Register = PR is 12 bits = 000000000000.

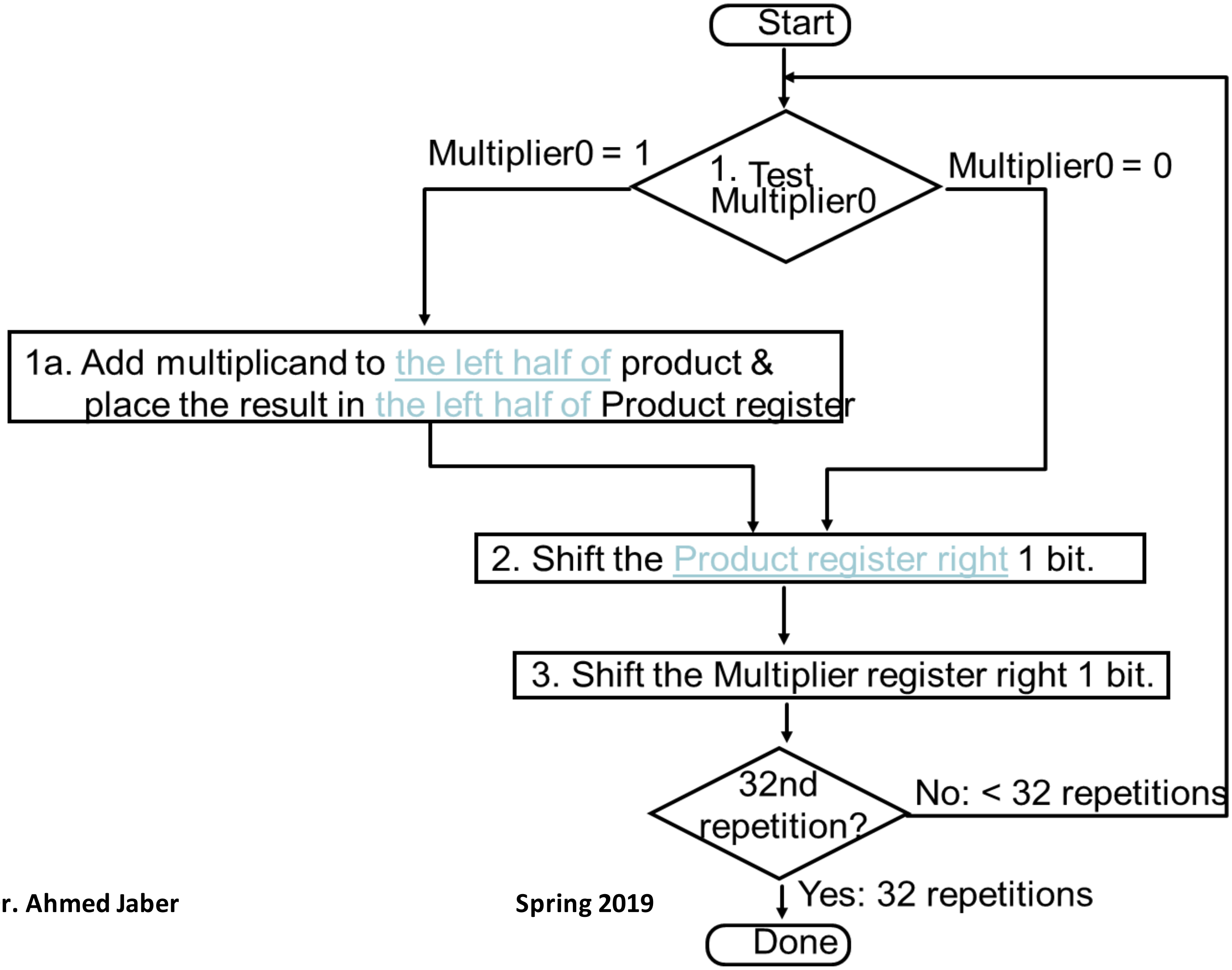


	6-bit x 6-bit 1st Version multiplier (58 × 23)			
	Steps	MR	MC	PR
0	Initial Values	010111	000000111010	000000000000
1	1: MR [0] =1 -> PR=PR+MC 2: SH_R MR, SH_L MC 1-bit	001011	000001110100	000000111010
2	1: MR[0] =1 -> PR=PR+MC 2: SH_R MR, SH_L MC 1-bit	000101	000011101000	000010101110
3	1: MR[0] =1 -> PR=PR+MC 2: SH_R MR, SH_L MC 1-bit	000010	000111010000	000110010110
4	1: MR[0] =0 -> SH_R MR, SH_L MC 1-bit	000001	001110100000	000110010110
5	1: MR[0] =1 -> PR=PR+MC 2: SH_R MR, SH_L MC 1-bit	000000	011101000000	010100110110
6	1: MR[0] =0 -> SH_R MR, SH_L MC 1-bit	000000	111010000000	010100110110
Stop	Result : PR= 58 × 23 = 1334 = 0x536= $(010100110110)_2$			

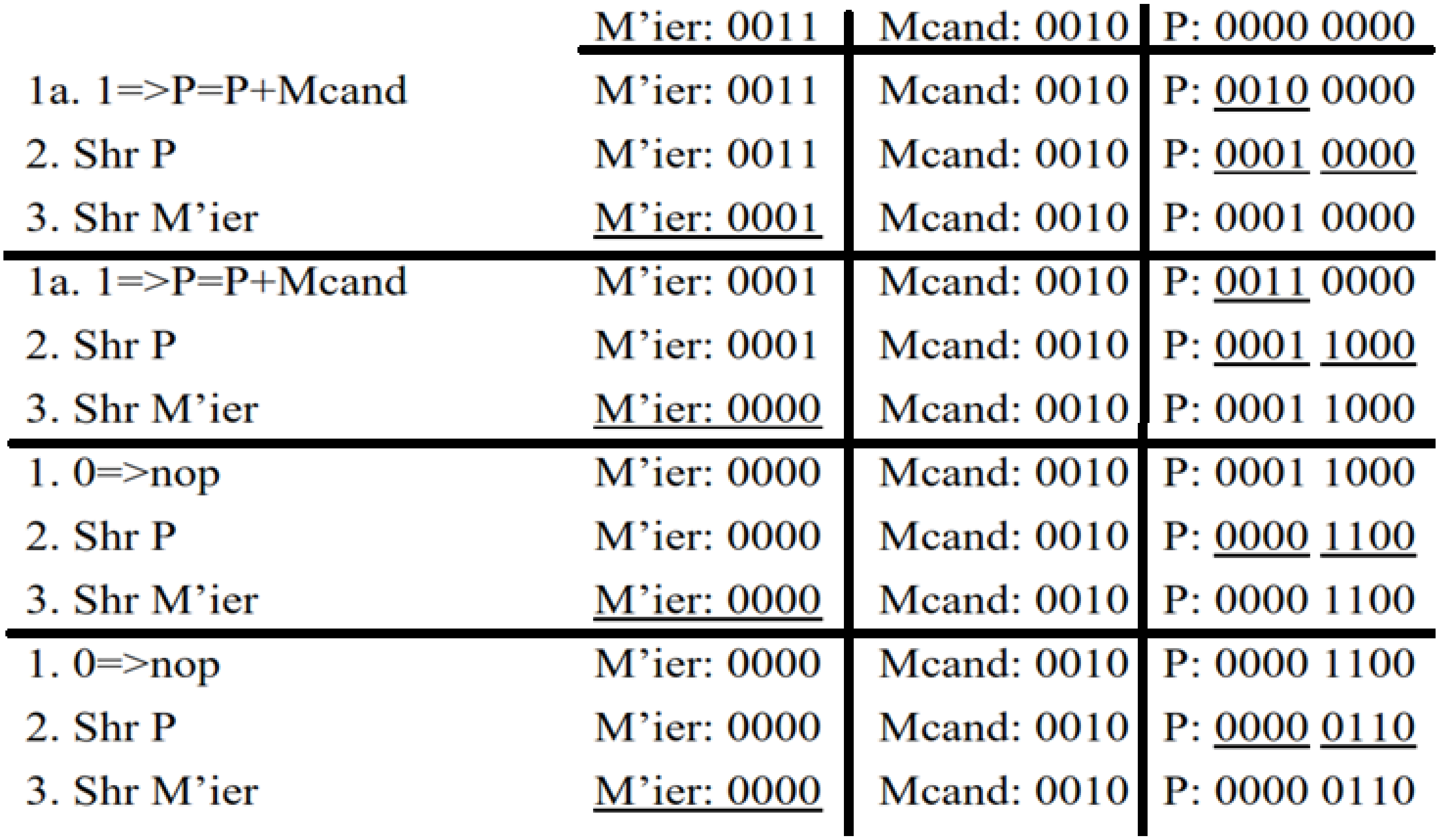


Multiply Hardware - Version 2

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg



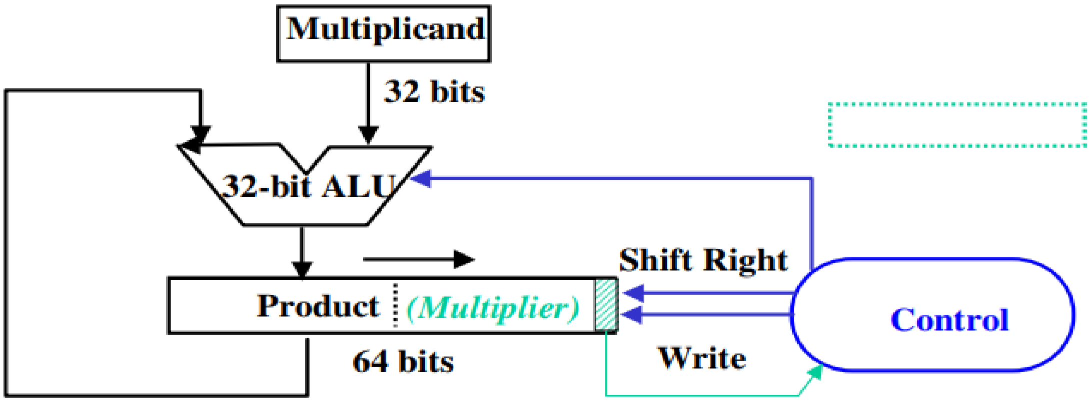
Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to the left half of product & place the result in the left half of Product register
2. Shift the Product register right 1 bit.
3. Shift the Multiplier register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



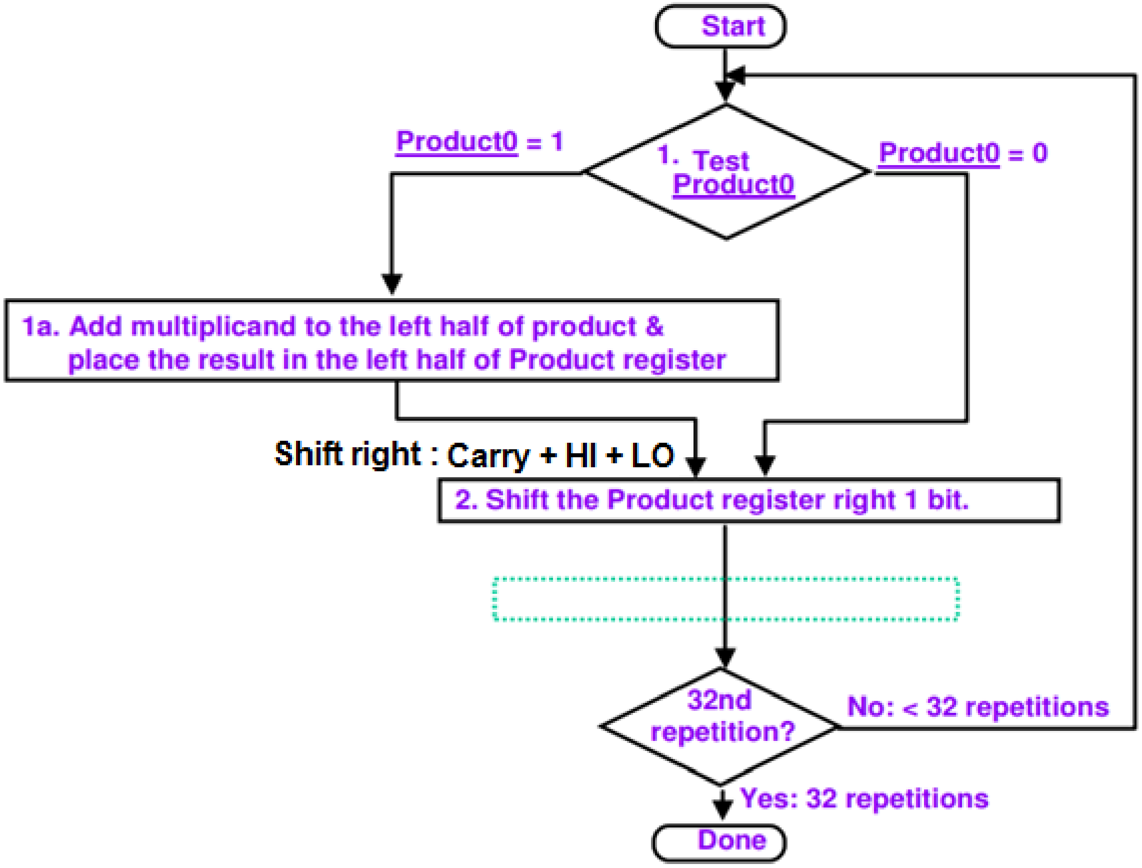
	M’ier: 0011	Mcand: 0010	P: 0000 0000
1a. 1=>P=P+Mcand	M’ier: 0011	Mcand: 0010	P: 0010 0000
2. Shr P	M’ier: 0011	Mcand: 0010	P: 0001 0000
3. Shr M’ier	M’ier: 0001	Mcand: 0010	P: 0001 0000
1a. 1=>P=P+Mcand	M’ier: 0001	Mcand: 0010	P: 0011 0000
2. Shr P	M’ier: 0001	Mcand: 0010	P: 0001 1000
3. Shr M’ier	M’ier: 0000	Mcand: 0010	P: 0001 1000
1. 0=>nop	M’ier: 0000	Mcand: 0010	P: 0001 1000
2. Shr P	M’ier: 0000	Mcand: 0010	P: 0000 1100
3. Shr M’ier	M’ier: 0000	Mcand: 0010	P: 0000 1100
1. 0=>nop	M’ier: 0000	Mcand: 0010	P: 0000 1100
2. Shr P	M’ier: 0000	Mcand: 0010	P: 0000 0110
3. Shr M’ier	M’ier: 0000	Mcand: 0010	P: 0000 0110

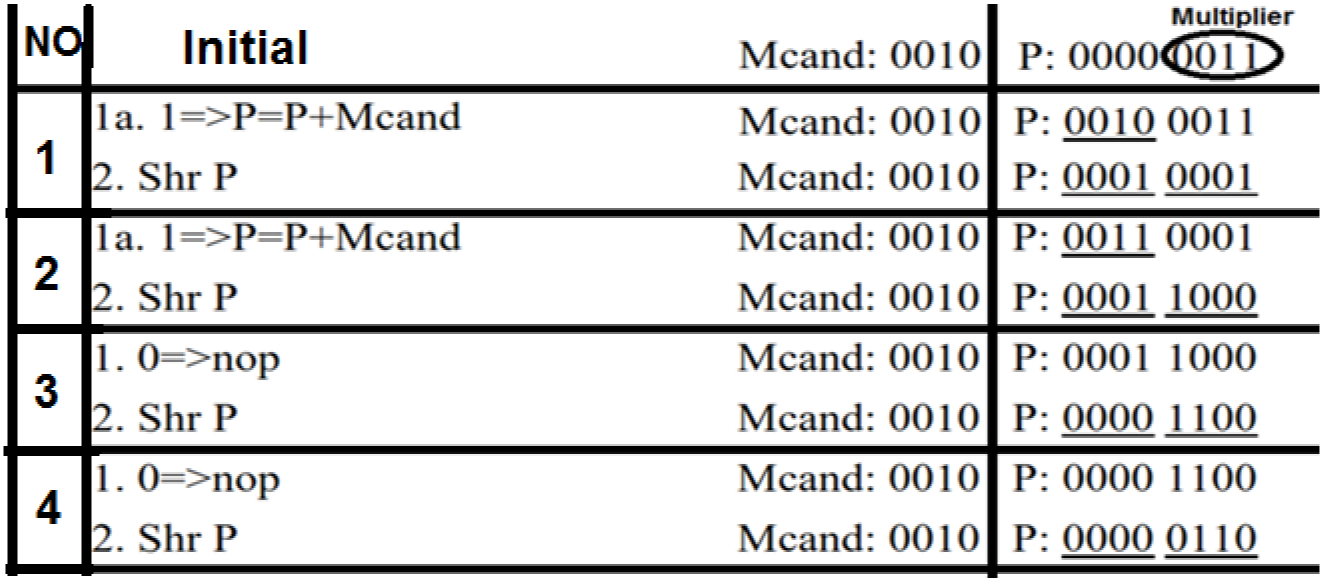
Multiply Hardware - Version 3

- Product register wastes space that exactly matches size of multiplier
  → combine Multiplier register and Product register
- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)



Multiply Algorithm - Version 3

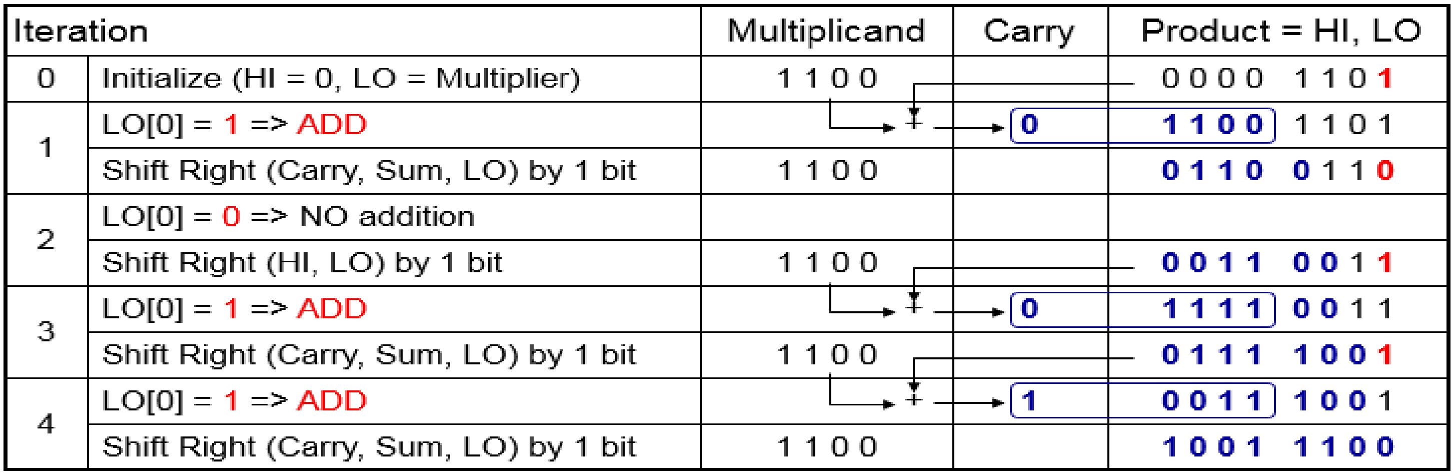

Start
Product0 = 1
1. Test Product0
Product0 = 0
1a. Add multiplicand to the left half of product & place the result in the left half of Product register
Shift right : Carry + HI + LO
2. Shift the Product register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done


Multiplier

NO	Initial	Mcand: 0010	P: 0000 0011
1	1a. 1=>P=P+Mcand	Mcand: 0010	P: 0010 0011
	2. Shr P	Mcand: 0010	P: 0001 0001
2	1a. 1=>P=P+Mcand	Mcand: 0010	P: 0011 0001
	2. Shr P	Mcand: 0010	P: 0001 1000
3	1. 0=>nop	Mcand: 0010	P: 0001 1000
	2. Shr P	Mcand: 0010	P: 0000 1100
4	1. 0=>nop	Mcand: 0010	P: 0000 1100
	2. Shr P	Mcand: 0010	P: 0000 0110

Example

- Consider: 1100_2 × 1101_2 , Product = 10011100_2
- 4-bit multiplicand and multiplier are used in this example
- 4-bit adder produces a **4-bit Sum + Carry bit**

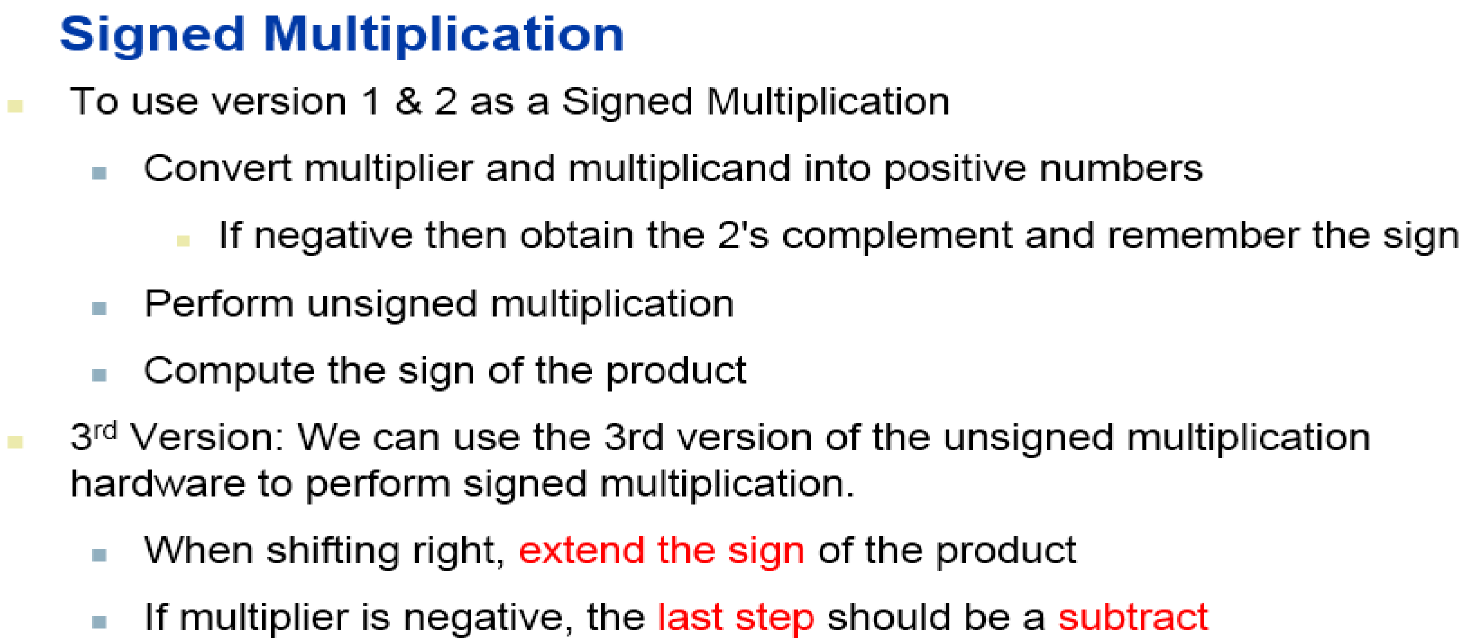


Iteration		Multiplicand	Carry	Product = HI, LO
0	Initialize (HI = 0, LO = Multiplier)	1 1 0 0		0 0 0 0 1 1 0 1
1	LO[0] = 1 => ADD		0	1 1 0 0 1 1 0 1
	Shift Right (Carry, Sum, LO) by 1 bit	1 1 0 0		0 1 1 0 0 1 1 0
2	LO[0] = 0 => NO addition			
	Shift Right (HI, LO) by 1 bit	1 1 0 0		0 0 1 1 0 0 1 1
3	LO[0] = 1 => ADD		0	1 1 1 1 0 0 1 1
	Shift Right (Carry, Sum, LO) by 1 bit	1 1 0 0		0 1 1 1 1 0 0 1
4	LO[0] = 1 => ADD		1	0 0 1 1 1 0 0 1
	Shift Right (Carry, Sum, LO) by 1 bit	1 1 0 0		1 0 0 1 1 1 0 0

Observations on Multiply Version 3

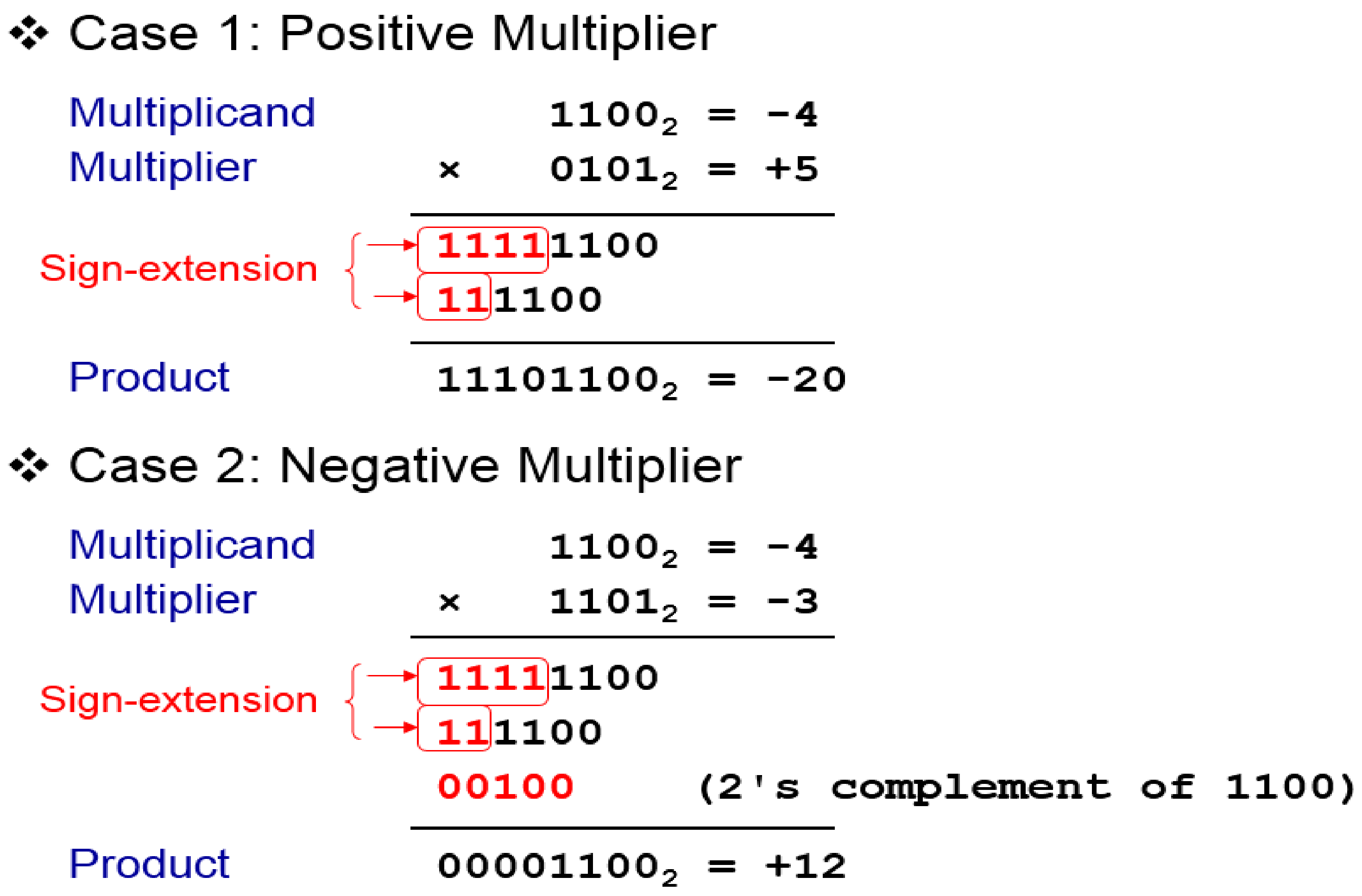
- 2 steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU

What about signed multiplication?



Signed Multiplication

- To use version 1 & 2 as a Signed Multiplication
  - Convert multiplier and multiplicand into positive numbers
    - If negative then obtain the 2's complement and remember the sign
  - Perform unsigned multiplication
  - Compute the sign of the product
- 3rd Version: We can use the 3rd version of the unsigned multiplication hardware to perform signed multiplication.
  - When shifting right, extend the sign of the product
  - If multiplier is negative, the last step should be a subtract



Case 1: Positive Multiplier

Multiplicand $1100_2 = -4$

Multiplier $\times \quad 0101_2 = +5$

Sign-extension

$\boxed{1111}1100$

$\boxed{11}1100$

Product $11101100_2 = -20$

Case 2: Negative Multiplier

Multiplicand $1100_2 = -4$

Multiplier $\times \quad 1101_2 = -3$

Sign-extension

$\boxed{1111}1100$

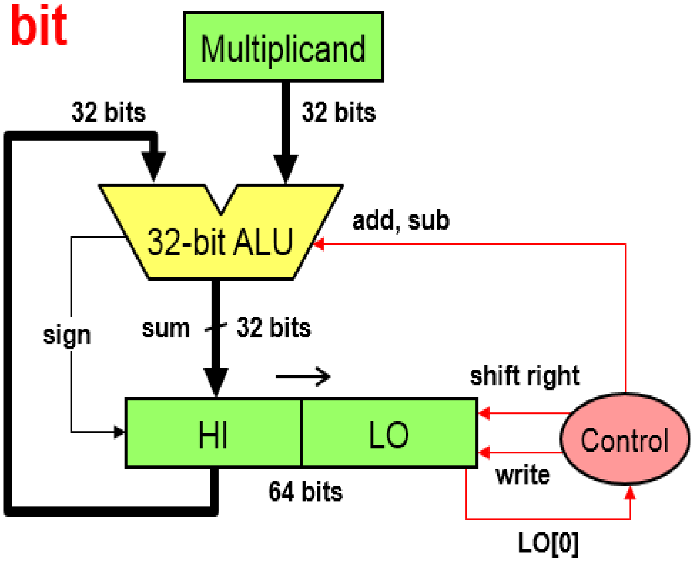
$\boxed{11}1100$

00100 (2's complement of 1100)

Product $00001100_2 = +12$

Sequential Signed Multiplier

- ALU produces: **32-bit sum + sign bit**
- Sign bit can be computed:
  - **No overflow: sign = sum[31]**
  - **If Overflow: sign = ~sum[31]**

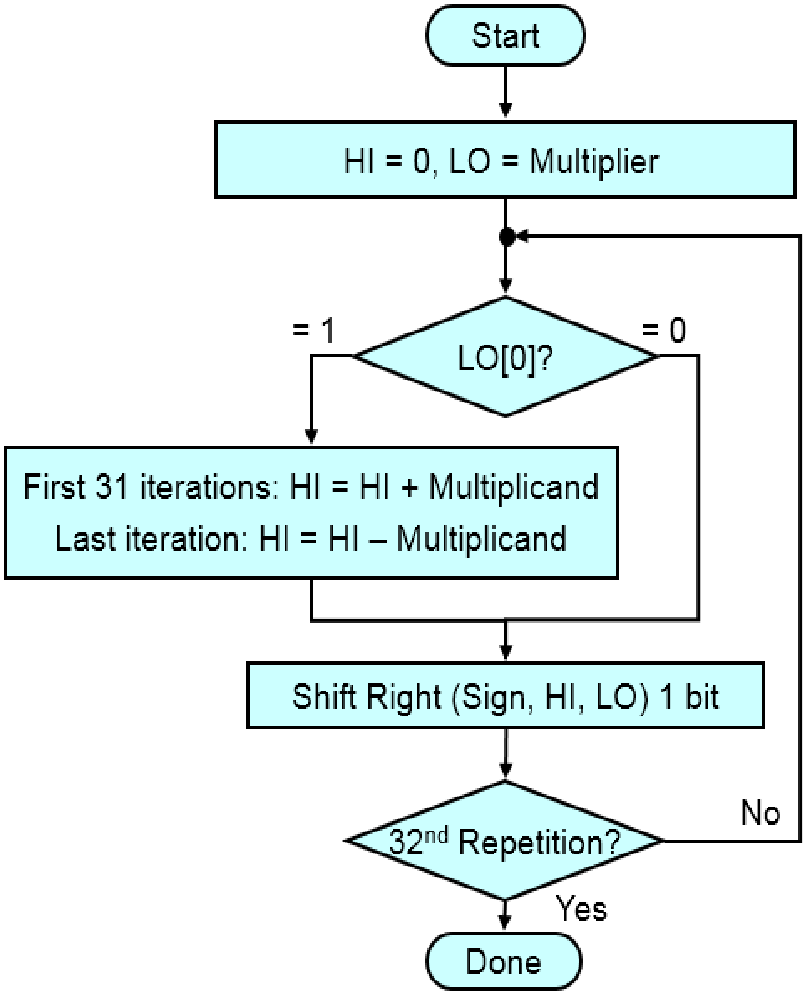


Positive
+ Positive

Overflow Negative (sign : 1)
Inverse sign : 0

Ngative
+ Negative

Overflow Positive (sign : 0)
Inverse sign : 1


Start
HI = 0, LO = Multiplier
LO[0]?
= 1
= 0
First 31 iterations: HI = HI + Multiplicand
Last iteration: HI = HI – Multiplicand
Shift Right (Sign, HI, LO) 1 bit
32nd Repetition?
No
Yes
Done

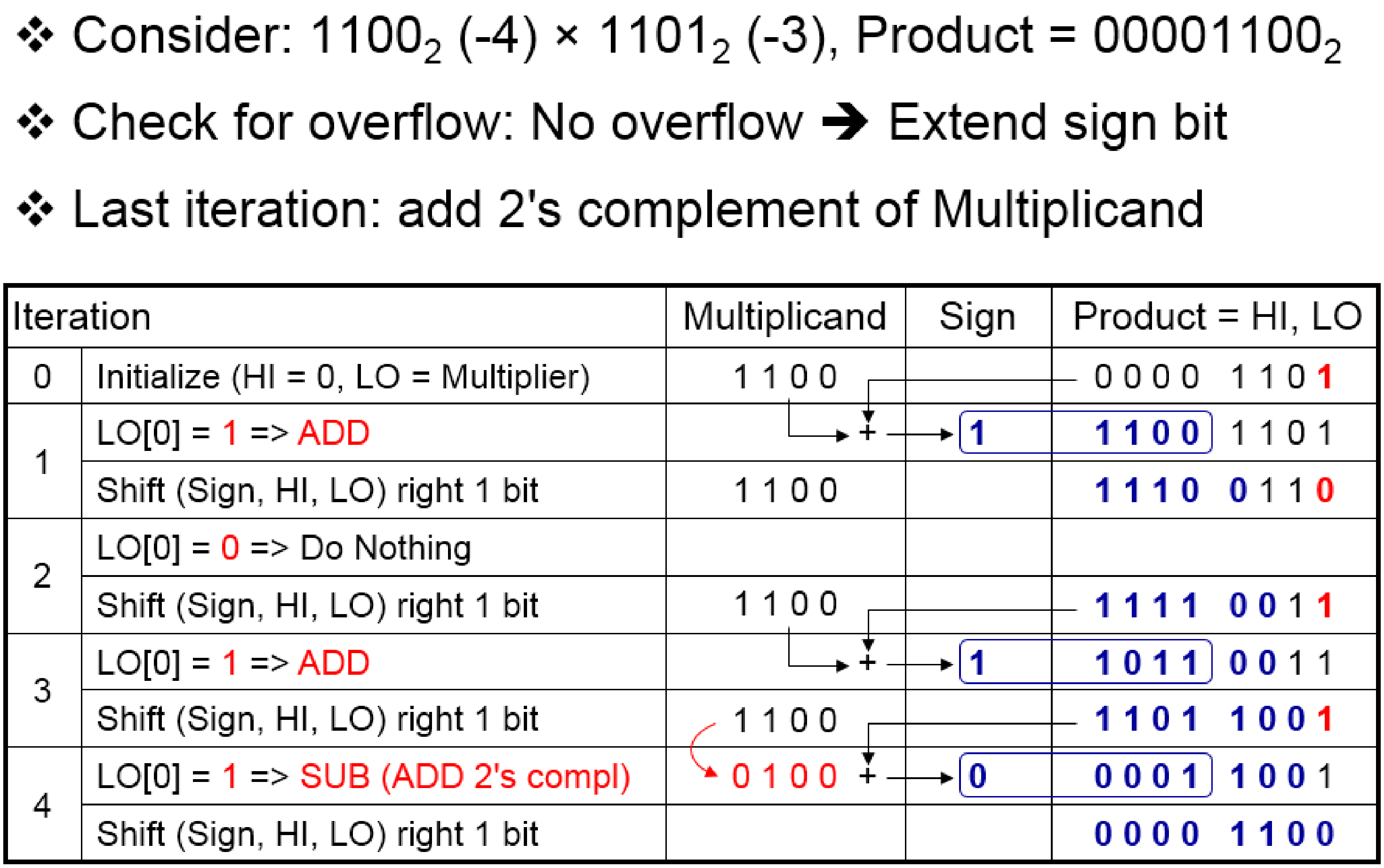
Example: 3rd Version Signed Multiplication

- **Multiplicand = -30 = signed 6-bit = $(100010)_2$**
- **Multiplier = 15 = signed 6-bit = $(001111)_2$**
- **Product = -30 × 15 = - 450 = $(111000111110)_2$**
- **Initial Values:**
- **Multiplicand Register = MC is 6 bits = 100010.**
- **Product Register {HI,LO} 12-bit**

Iteration	3rd Version signed Multiplier − 30 × 15 , MC = 100010			
	Step	Sign	HI	LO
0	Initial Values	0	000000	001111
1	1: PR[0] =1--> Hi=Hi+MC	1	100010	001111
	2: Shift right (HI,LO) 1 bit	1	110001	000111
2	1: PR[0] =1--> Hi=Hi+MC	1	010011	000111
	2: Shift right (HI,LO) 1 bit	1	101001	100011
3	1: PR[0] =1--> Hi=Hi+MC	1	001011	100011
	2: Shift right (HI,LO) 1 bit	1	100101	110001
4	1: PR[0] =1--> Hi=Hi+MC	1	000111	110001
	2: Shift right (HI,LO) 1 bit	1	100011	111000
5	1: PR[0] = 0-->	1	100011	111000
	Shift right (HI,LO) 1 bit	1	110001	111100

6	1: PR[0] = 0--> Shift right (HI,LO) 1 bit	1 110001 111100 1 111000 111110
Stop	Result : PR= -30 × 15 = -450 =	$(111000\ 111110)_2$

Example



- Consider: 1100_2 (-4) × 1101_2 (-3), Product = 00001100_2
- Check for overflow: No overflow ➔ Extend sign bit
- Last iteration: add 2's complement of Multiplicand

Iteration		Multiplicand	Sign	Product = HI, LO
0	Initialize (HI = 0, LO = Multiplier)	1 1 0 0		0 0 0 0 1 1 0 1
1	LO[0] = 1 => ADD	+	1	1 1 0 0 1 1 0 1
	Shift (Sign, HI, LO) right 1 bit	1 1 0 0		1 1 1 0 0 1 1 0
2	LO[0] = 0 => Do Nothing			
	Shift (Sign, HI, LO) right 1 bit	1 1 0 0		1 1 1 1 0 0 1 1
3	LO[0] = 1 => ADD	+	1	1 0 1 1 0 0 1 1
	Shift (Sign, HI, LO) right 1 bit	1 1 0 0		1 1 0 1 1 0 0 1
4	LO[0] = 1 => SUB (ADD 2's compl)	0 1 0 0 +	0	0 0 0 1 1 0 0 1
	Shift (Sign, HI, LO) right 1 bit			0 0 0 0 1 1 0 0

Booth’s Algorithm for signed multiplication

(This algorithm was invented by Andrew Donald Booth in 1950). Booth’s algorithm is a powerful algorithm that is used for signed multiplication. It generates a 2n bit product for two n bit signed numbers.

- In general, in the Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left

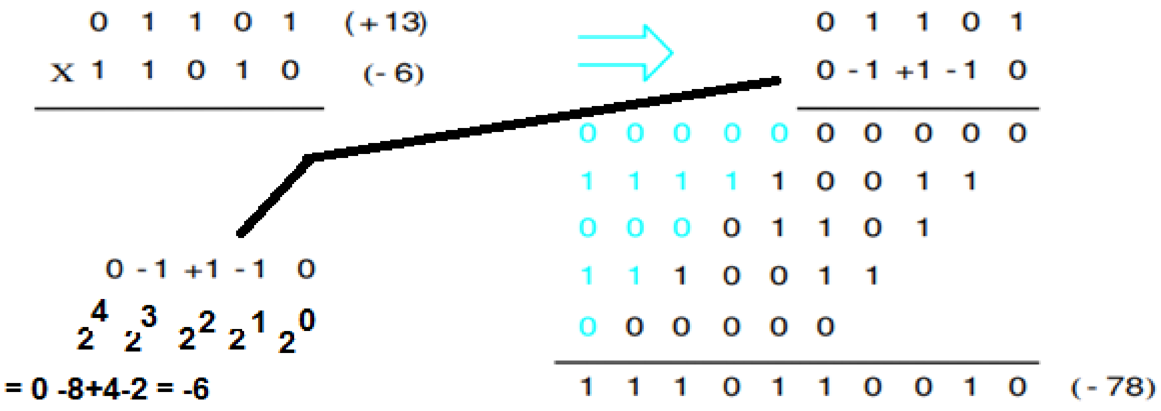
```
0  0  1  0  1  1  0  0  1   1  1  0  1  0  1  1  0  0
                          ⇓
0 +1 -1 +1  0 -1  0 +1  0   0 -1 +1 -1 +1  0 -1  0  0
```

Booth recoding of a multiplier

Booth Multiplier Recording Table

Multiplier		Version of multiplicand selected by bit
Bit *i*	Bit *i*-1	
0	0	0 X M
0	1	+1 X M

Booth Algorithm Example for Negative Multiplier

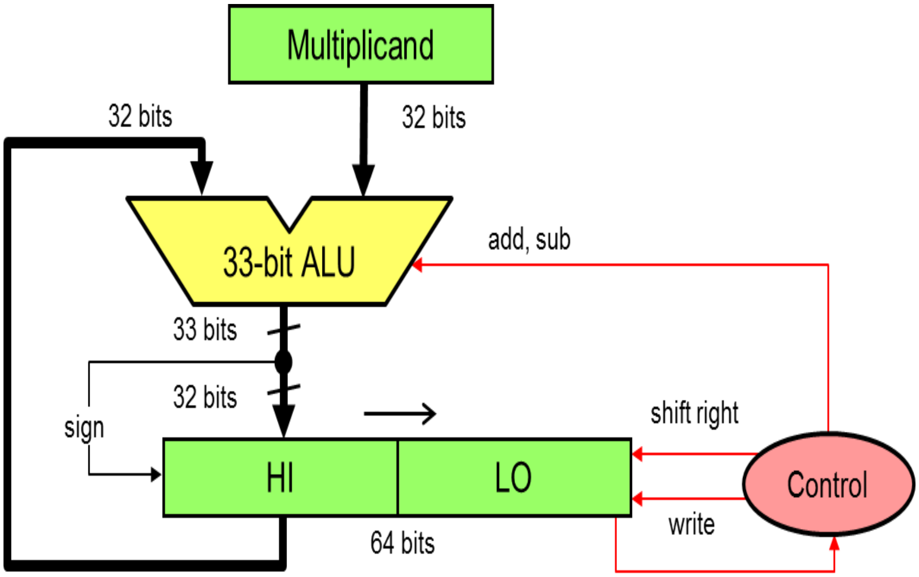


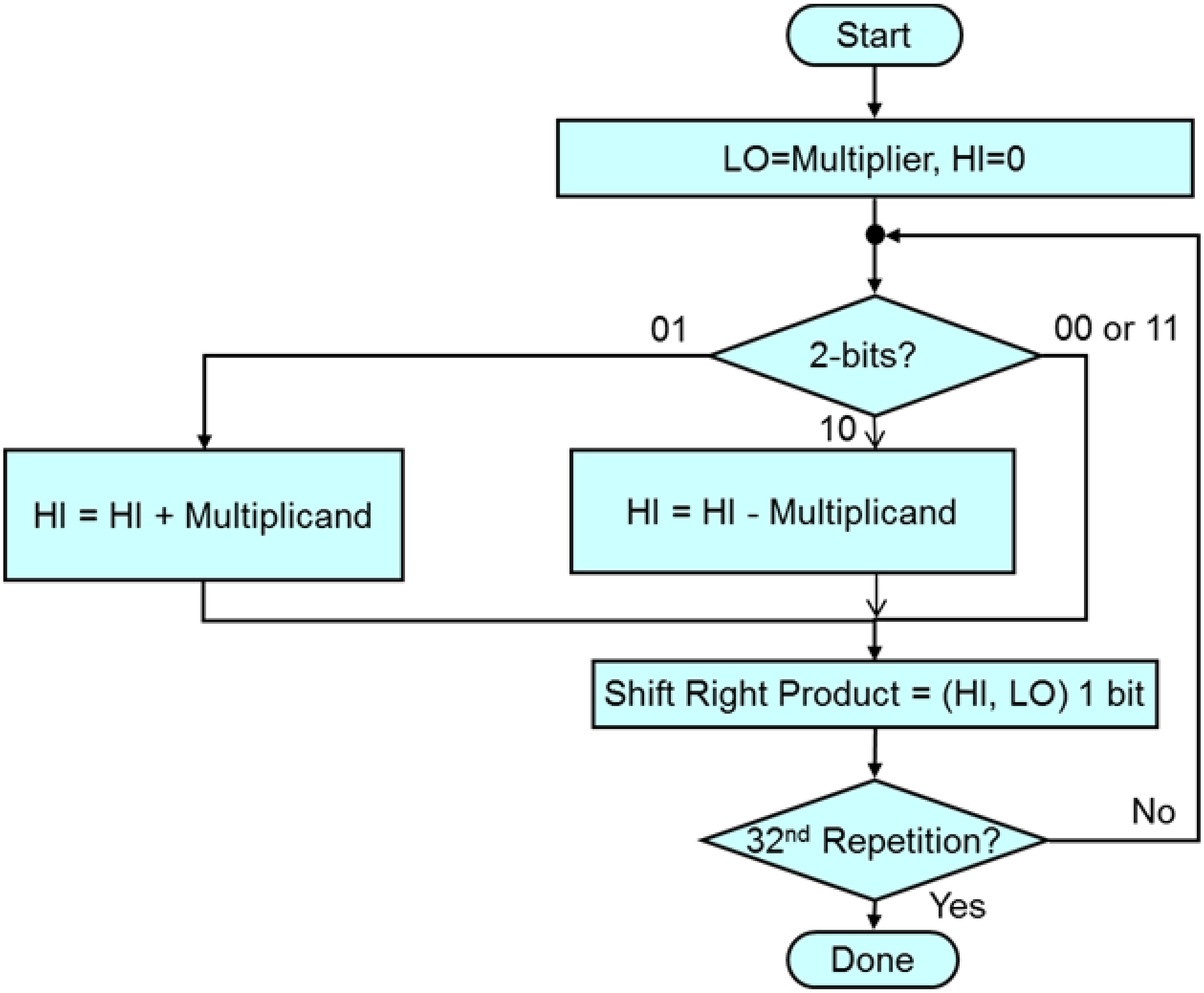
```
   0  1  1  0  1   (+13)
 x 1  1  0  1  0   (-6)
 ______________
   0 -1 +1 -1  0
```

$2^4\ 2^3\ 2^2\ 2^1\ 2^0$

= 0 -8+4-2 = -6

```
                      0  1  1  0  1
                      0 -1 +1 -1  0
                     ______________
 0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  0  0  1  1
 0  0  0  0  1  1  0  1
 1  1  1  0  0  1  1
 0  0  0  0  0  0
______________________________
 1  1  1  0  1  1  0  0  1  0   (-78)
```


Multiplicand
32 bits
32 bits
33-bit ALU
add, sub
33 bits
32 bits
sign
shift right
HI
LO
Control
write
64 bits


Start
LO=Multiplier, HI=0
2-bits?
01
00 or 11
10
HI = HI + Multiplicand
HI = HI - Multiplicand
Shift Right Product = (HI, LO) 1 bit
32nd Repetition?
No
Yes
Done

Example: (Multiplicand is negative)

Iter-ation	Booth's Algorithm Multiplier − 30 × 15 = -450 MC = 30_{10} = 100010_2		
	Step	(Sign , HI LO,	Initial)
0	Initial Values		
	LO of pruduct = MR	0 000000 001111	0
1	1: {PR[0], 0} = 10		
	Subtract (HI=HI-MC)	0 011110 001111	0
	2: Shift right PR	0 001111 000111	1
2	1: {PR[0], 1} = 11		
	Shift right	0 000111 100011	1
3	1: {PR[0], 1} = 11		
	Shift right	0 000011 110001	1
4	1: {PR[0], 1} = 11		
	Shift right	0 000001 111000	1
5	1: {PR[0], 1} = 01		
	ADD (HI=HI+MC)	1 100011 111000	1
	2: Shift right PR	1 110001 111100	0
6	1: {PR[0], 0} = 00		
	Shift right	1 111000 111110	0
STOP	Result : PR= -30 × 15 = -450 = $(111000\,111110)_2$		

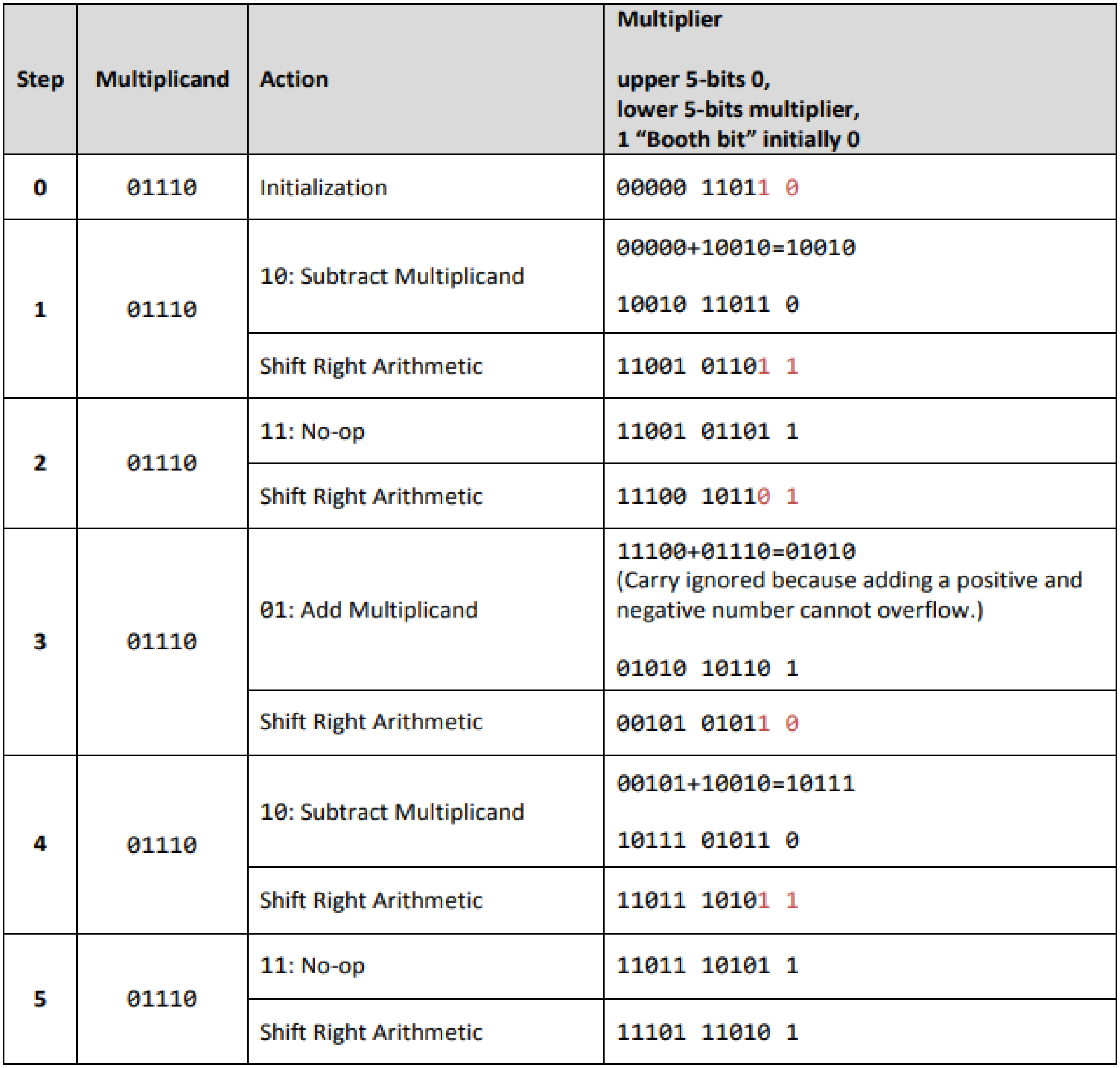
Example: (Multiplier is negative)

Multiply 14 times -5 using 5-bit numbers (10-bit result).

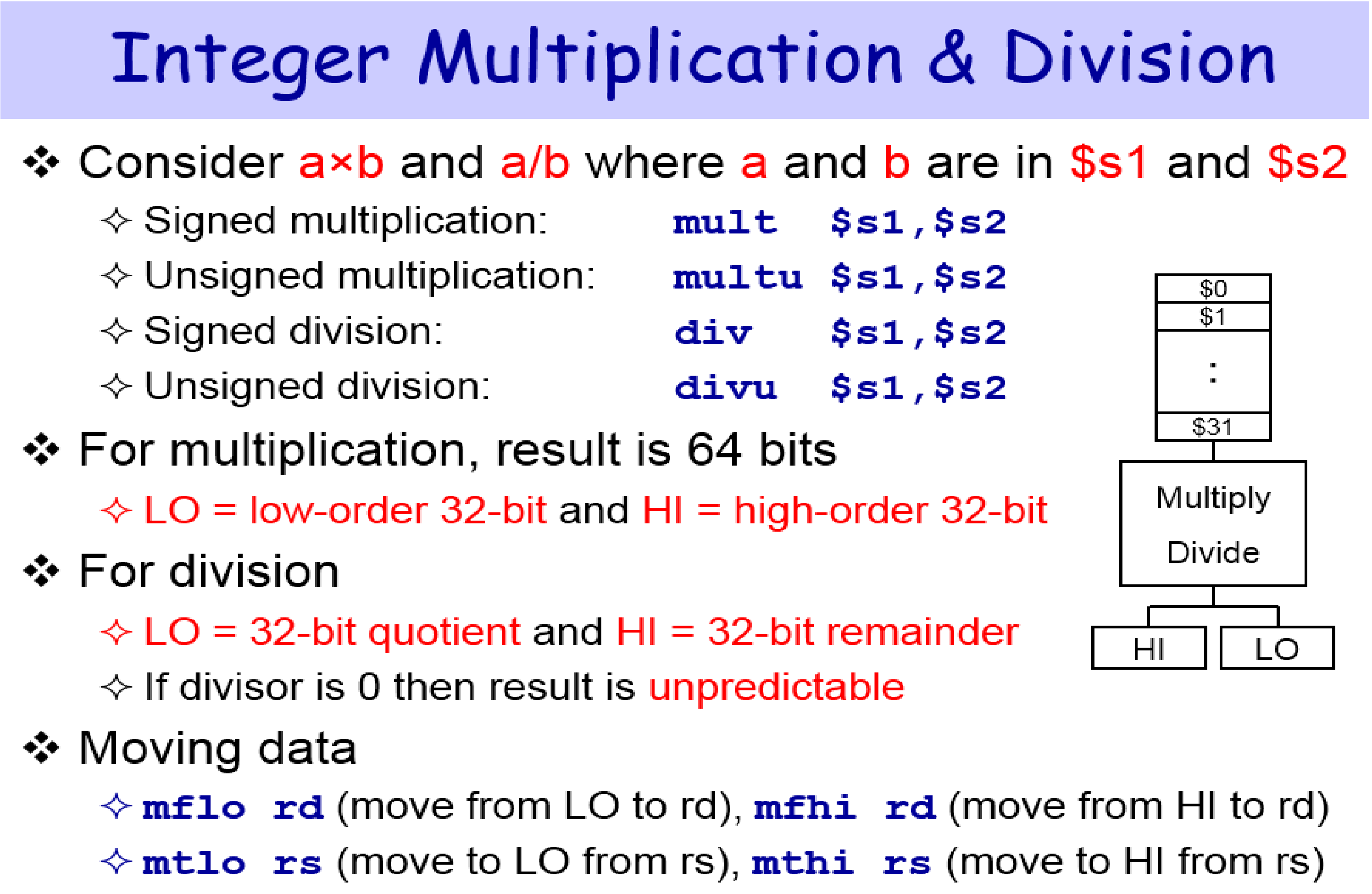
14 in binary: 01110 (Multiplicand)

-5 in binary: 11011 (Multiplier)

Expected result: 14 x -5 = -70 in binary: 11101 11010



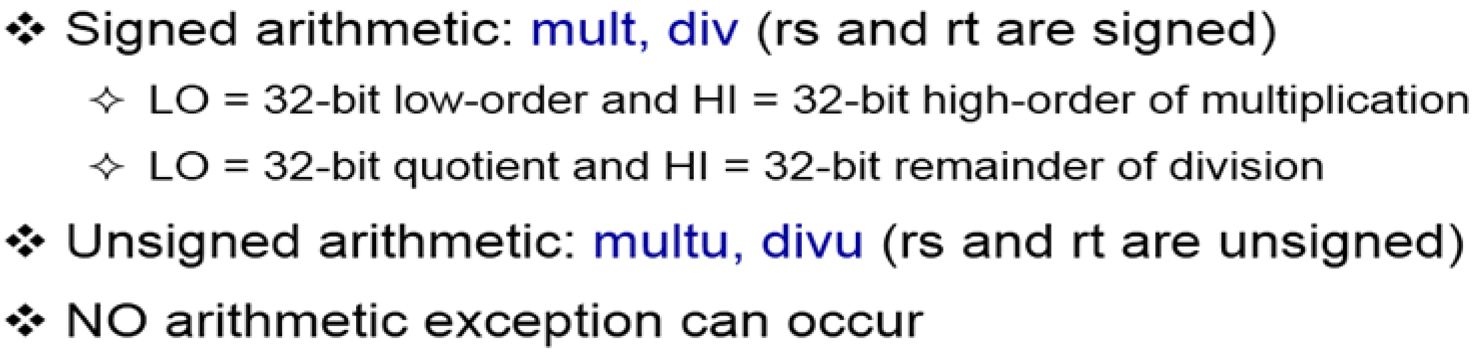
Step	Multiplicand	Action	Multiplier upper 5-bits 0, lower 5-bits multiplier, 1 “Booth bit” initially 0
0	01110	Initialization	00000 11011 0
1	01110	10: Subtract Multiplicand	00000+10010=10010 10010 11011 0
		Shift Right Arithmetic	11001 01101 1
2	01110	11: No-op	11001 01101 1
		Shift Right Arithmetic	11100 10110 1
3	01110	01: Add Multiplicand	11100+01110=01010 (Carry ignored because adding a positive and negative number cannot overflow.) 01010 10110 1
		Shift Right Arithmetic	00101 01011 0
4	01110	10: Subtract Multiplicand	00101+10010=10111 10111 01011 0
		Shift Right Arithmetic	11011 10101 1
5	01110	11: No-op	11011 10101 1
		Shift Right Arithmetic	11101 11010 1



Integer Multiplication & Division

- Consider a×b and a/b where a and b are in $s1 and $s2
  - Signed multiplication: `mult $s1,$s2`
  - Unsigned multiplication: `multu $s1,$s2`
  - Signed division: `div $s1,$s2`
  - Unsigned division: `divu $s1,$s2`
- For multiplication, result is 64 bits
  - LO = low-order 32-bit and HI = high-order 32-bit
- For division
  - LO = 32-bit quotient and HI = 32-bit remainder
  - If divisor is 0 then result is unpredictable
- Moving data
  - `mflo rd` (move from LO to rd), `mfhi rd` (move from HI to rd)
  - `mtlo rs` (move to LO from rs), `mthi rs` (move to HI from rs)

$0
$1
:
$31
Multiply
Divide
HI
LO

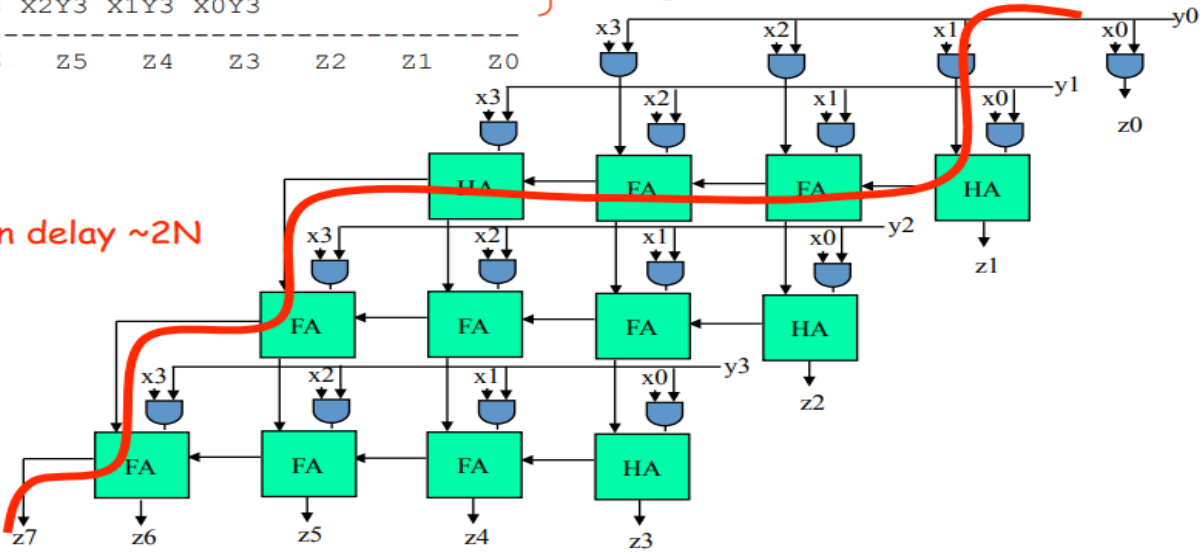


- Signed arithmetic: mult, div (rs and rt are signed)
  - LO = 32-bit low-order and HI = 32-bit high-order of multiplication
  - LO = 32-bit quotient and HI = 32-bit remainder of division
- Unsigned arithmetic: multu, divu (rs and rt are unsigned)
- NO arithmetic exception can occur

Combinational Multiplier (unsigned)

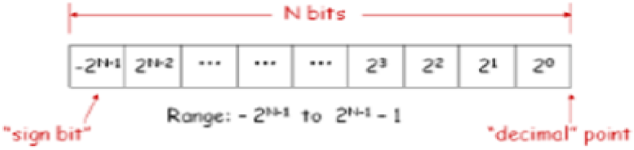
```
                                X3    X2    X1    X0   ← multiplicand
                             *  Y3    Y2    Y1    Y0   ← multiplier
                             --------------------
                              X3Y0  X2Y0  X1Y0  X0Y0 ┐
+                       X3Y1  X2Y1  X1Y1  X0Y1       │ Partial products, one for each bit in
+                 X3Y2  X2Y2  X1Y2  X0Y2             │ multiplier (each bit needs just one
+           X3Y3  X2Y3  X1Y3  X0Y3                   ┘ AND gate)
--------------------------------------------------
      Z7    Z6    Z5    Z4    Z3    Z2    Z1    Z0
```

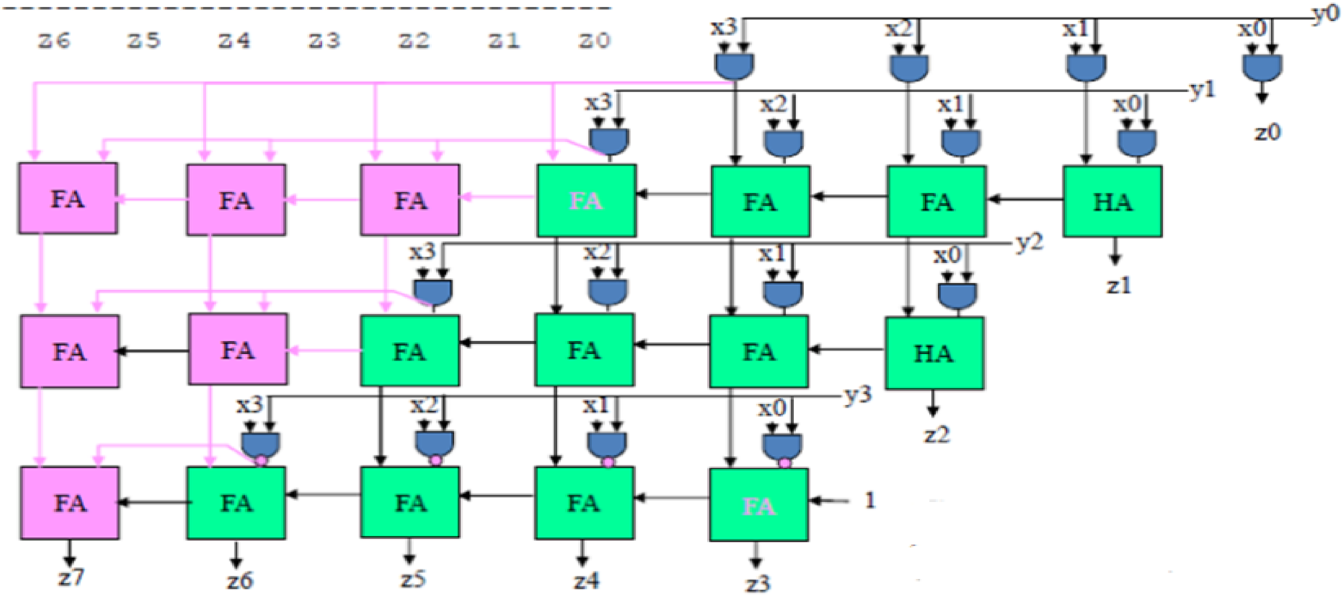
- Propagation delay ~2N



Combinational Multipliers (signed)

```
                         X3    X2    X1    X0
                       * Y3    Y2    Y1    Y0
                       --------------------
  X3Y0  X3Y0  X3Y0  X3Y0  X3Y0  X2Y0  X1Y0  X0Y0
+ X3Y1  X3Y1  X3Y1  X3Y1  X2Y1  X1Y1  X0Y1
+ X3Y2  X3Y2  X3Y2  X2Y2  X1Y2  X0Y2
- X3Y3  X3Y3  X2Y3  X1Y3  X0Y3
----------------------------------------
   Z7    Z6    Z5    Z4    Z3    Z2    Z1    Z0
```





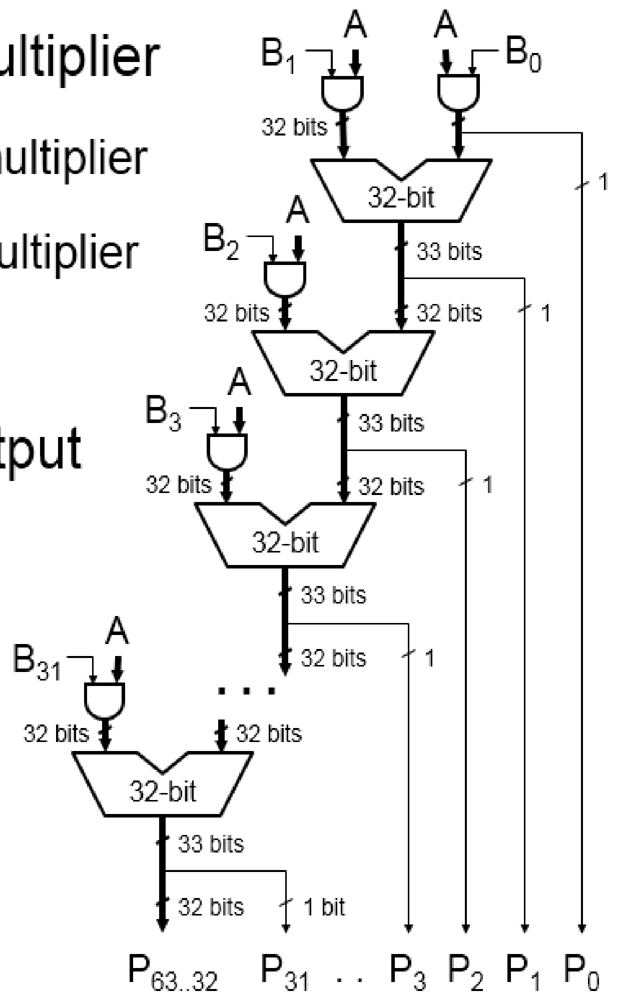
Faster Multiplier

- ❖ Moore’s Law has provided so much more in resources that hardware designers can now build much faster multiplication hardware.

- ❖ Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.

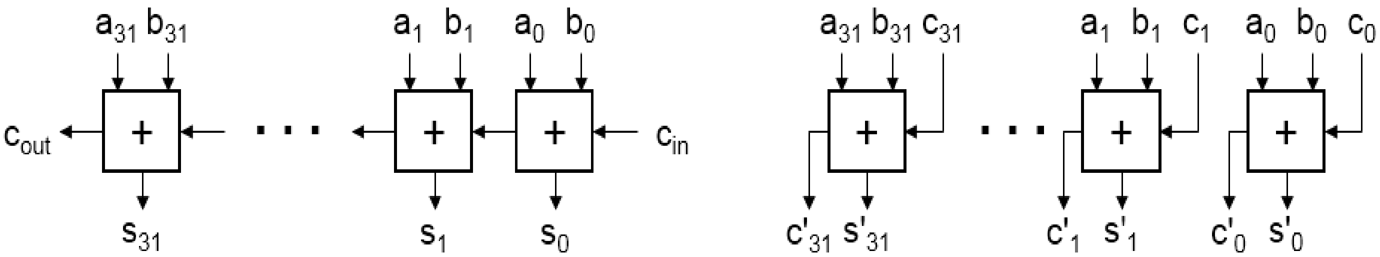
- ❖ Fast multiplication hardware: Rather than use a single 32 bit adder 31 times, the following hardware unrolls the loop to use 31 adders.

- ❖ 32-bit adder for each bit of the multiplier
  - ✧ 31 adders are needed for a 32-bit multiplier
  - ✧ AND multiplicand with each bit of multiplier
  - ✧ Product = accumulated shifted sum
- ❖ Each adder produces a 33-bit output
  - ✧ Most significant bit is a carry bit
  - ✧ Least significant bit is a product bit
  - ✧ Upper 32 bits go to next adder
- ❖ Array multiplier can be optimized
  - ✧ **Carry save adders** reduce delays



Carry Save Adders

- A n-bit carry-save adder produces two n-bit outputs
  - n-bit partial sum bits and n-bit carry bits
- All the n bits of a carry-save adder work in parallel
  - The carry does not propagate as in a carry-propagate adder
  - This is why a carry-save is faster than a carry-propagate adder
- Useful when adding multiple numbers (as in multipliers)

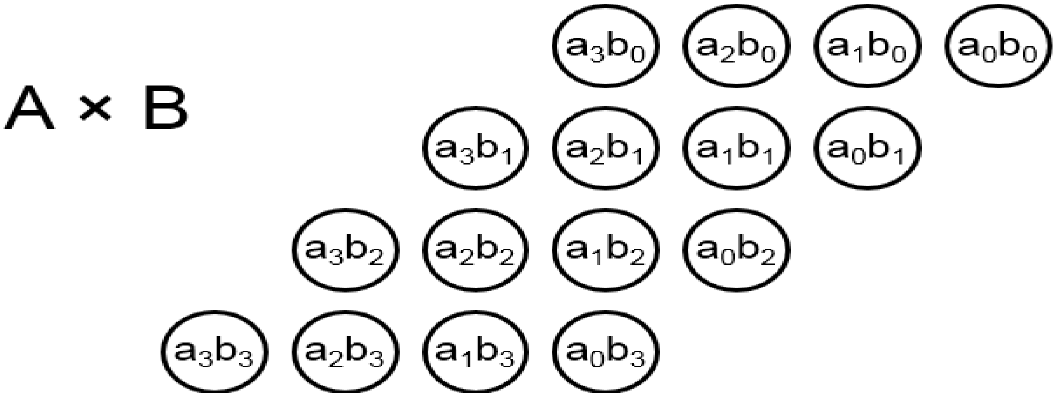


Carry-Propagate Adder

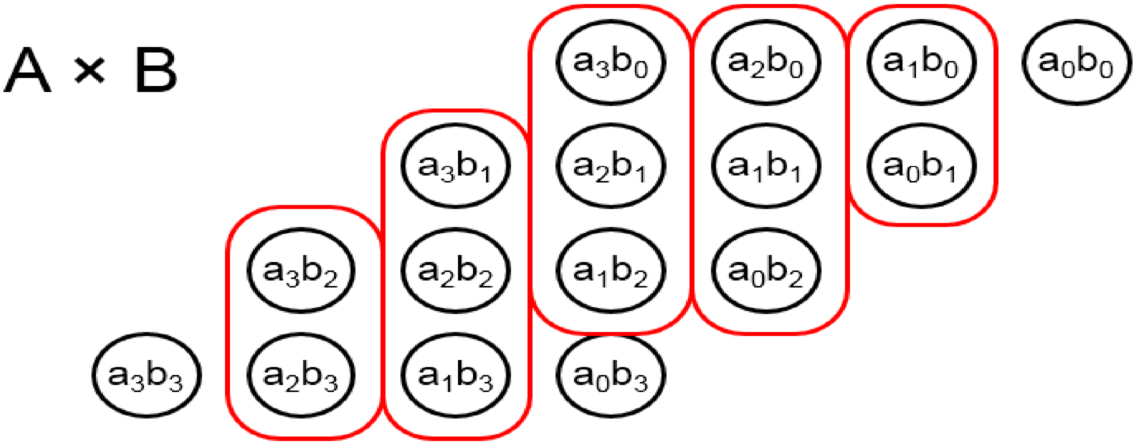
Carry-Save Adder

Carry-Save Adders in a Multiplier

- Suppose we want to multiply two numbers A and B
  - Example on 4-bit numbers: $A = a_3\ a_2\ a_1\ a_0$ and $B = b_3\ b_2\ b_1\ b_0$
- **Step 1**: AND (multiply) each bit of A with each bit of B
  - Requires n^2 AND gates and produces n^2 product bits



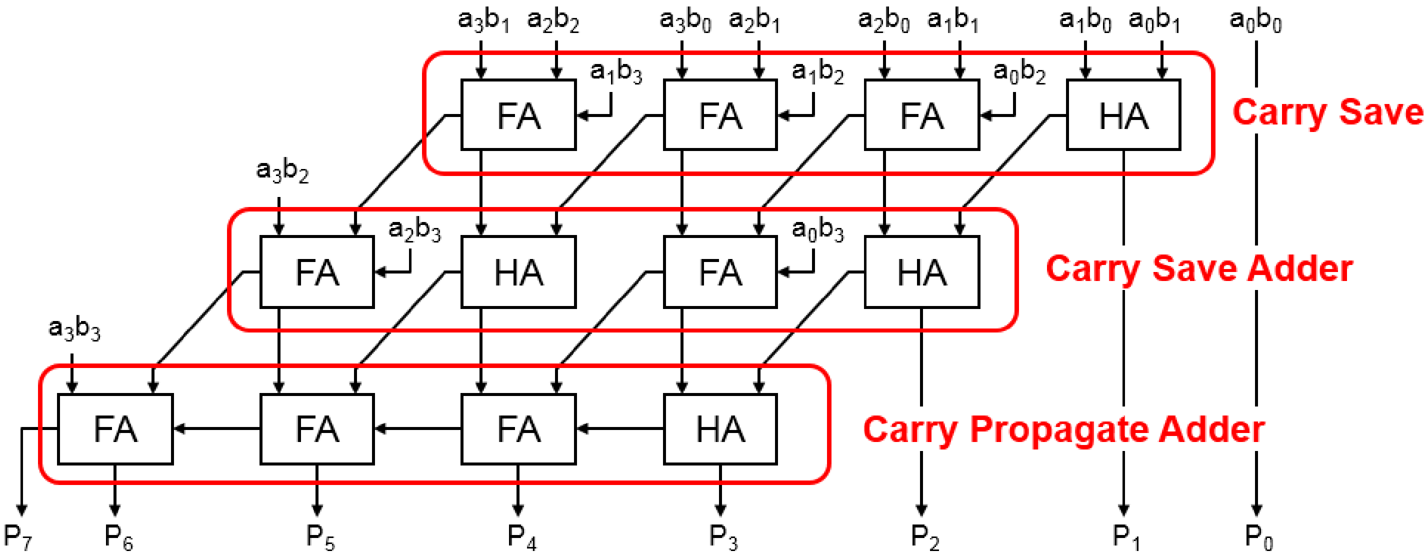
- **Step 2**
- ADD the product bits vertically using **Carry-Save adders**
  - Full Adder adds three vertical bits
  - Half Adder adds two vertical bits
  - Each adder produces a partial sum and a carry
- Use **Carry-propagate adder** for final addition



Step 3: Use **carry save adders** to add the partial products

- Reduce the partial products to just two numbers

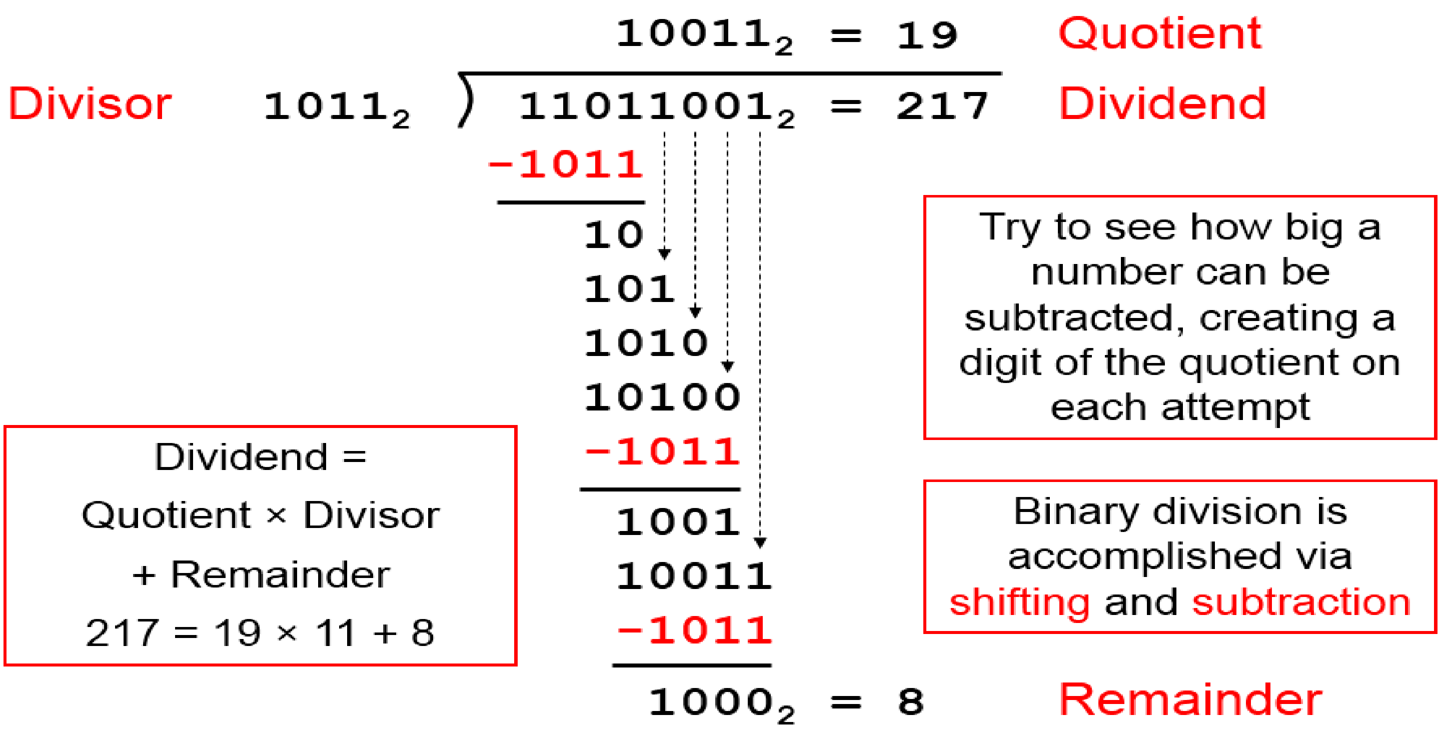
Step 4: Use **carry-propagate adder** to add last two numbers



Summary of a Fast Multiplier

- A fast n-bit × n-bit multiplier requires:
  - n^2 AND gates to produce n^2 product bits in parallel
  - Many adders to perform additions in parallel
- Uses carry-save adders to reduce delays
- Higher cost (more chip area) than sequential multiplier
- Higher performance (faster) than sequential multiplier

Unsigned Division



```
                    10011₂ = 19      Quotient
Divisor   1011₂ ) 11011001₂ = 217    Dividend
                 -1011
                  ----
                     10
                     101
                     1010
                     10100
                     -1011
                     -----
                      1001
                      10011
                      -1011
                      -----
                       1000₂ = 8     Remainder
```

Try to see how big a number can be subtracted, creating a digit of the quotient on each attempt

Dividend = Quotient × Divisor + Remainder

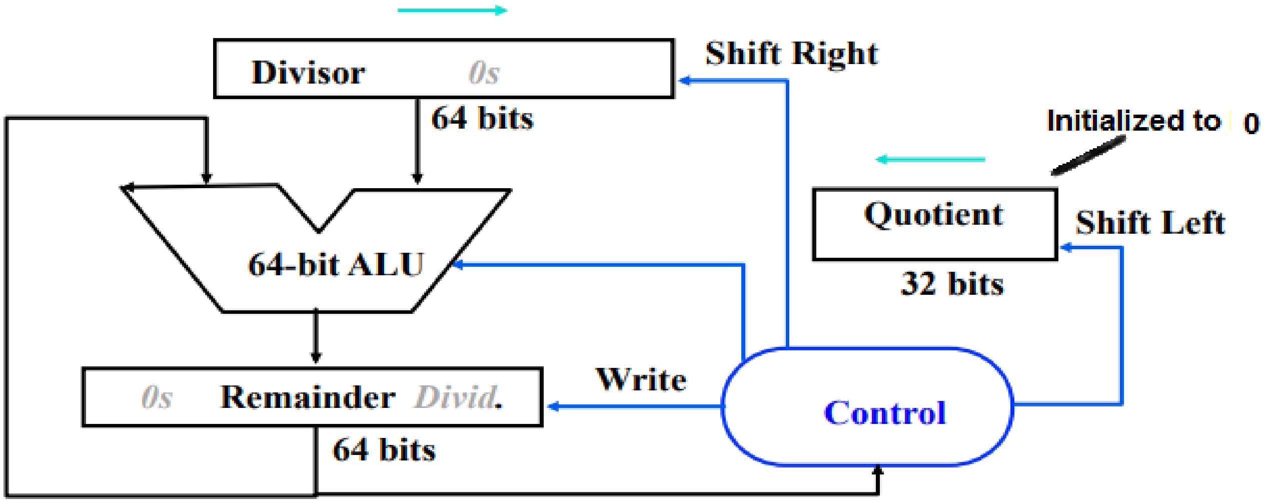
217 = 19 × 11 + 8

Binary division is accomplished via shifting and subtraction

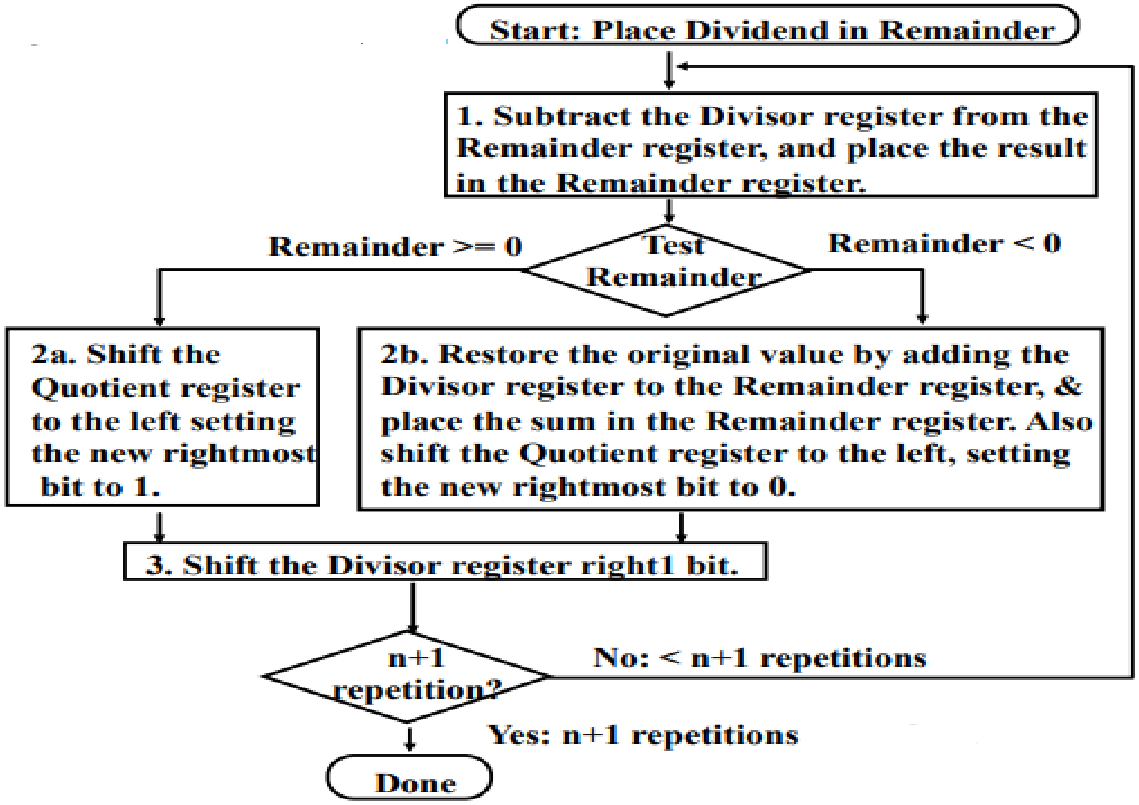
3 versions of divide, successive refinement

DIVIDE HARDWARE Version 1

- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



- Takes n+1 steps for n-bit Quotient & Rem.



Start: Place Dividend in Remainder

1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.

Test Remainder

Remainder >= 0

Remainder < 0

2a. Shift the Quotient register to the left setting the new rightmost bit to 1.

2b. Restore the original value by adding the Divisor register to the Remainder register, & place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new rightmost bit to 0.

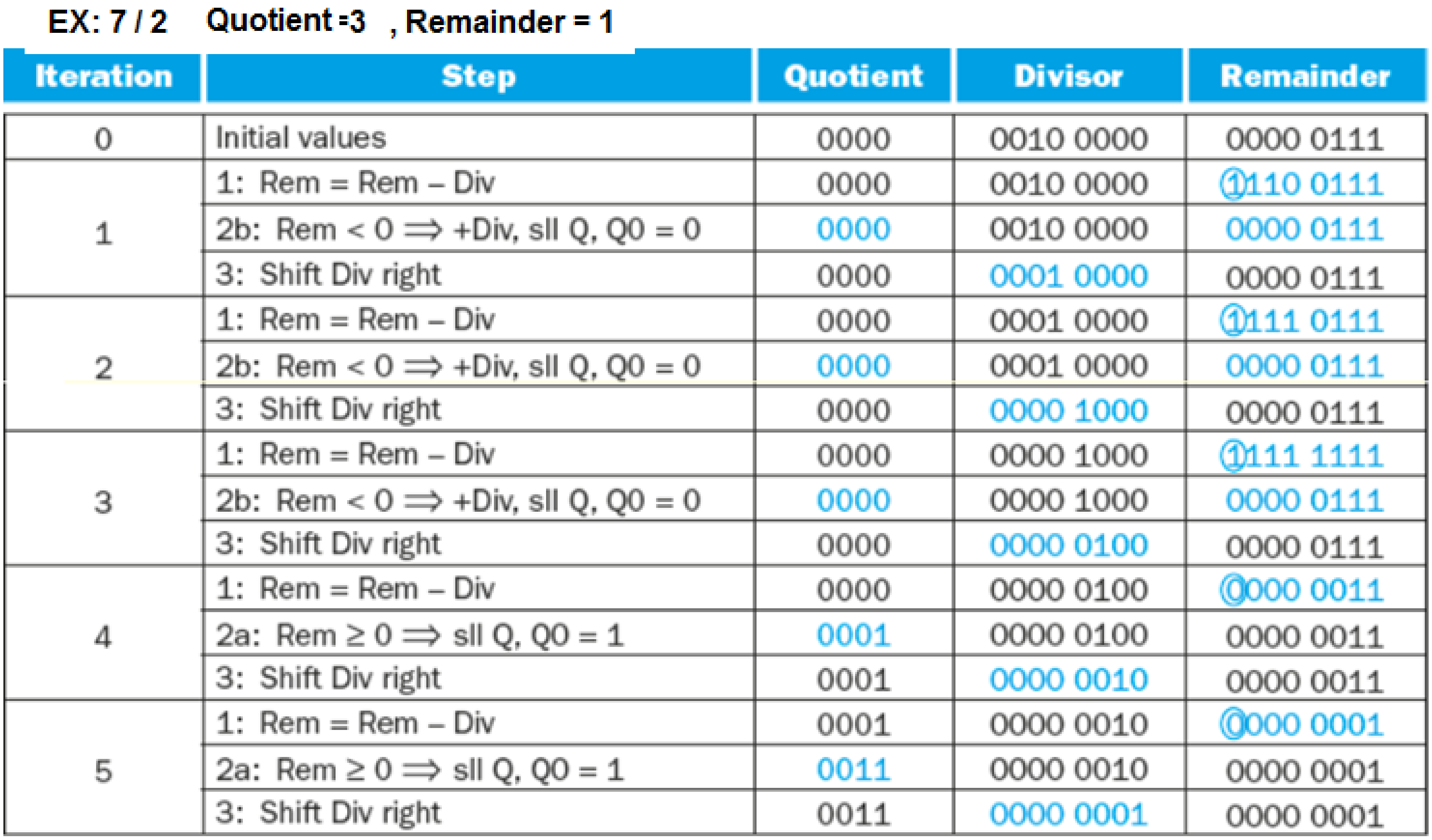
3. Shift the Divisor register right1 bit.

n+1 repetition?

No: < n+1 repetitions

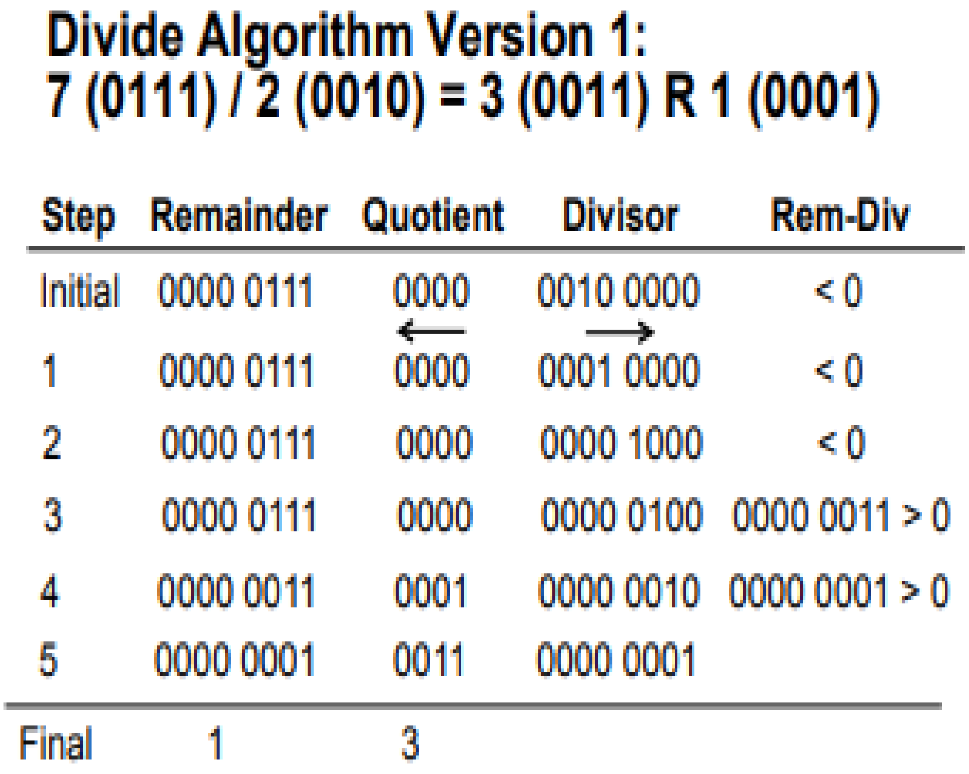
Yes: n+1 repetitions

Done



EX: 7 / 2 Quotient =3 , Remainder = 1

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001



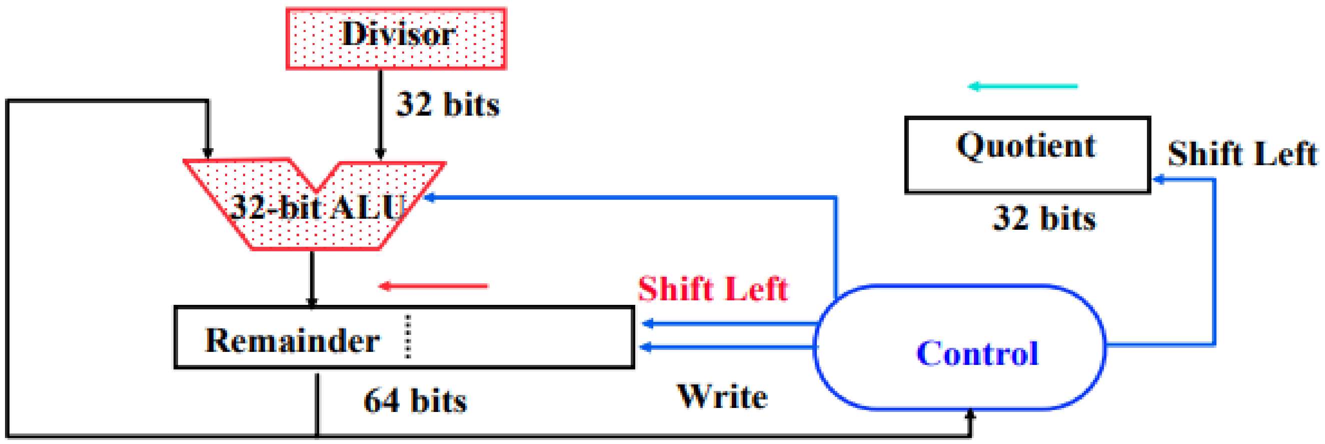
Divide Algorithm Version 1:
7 (0111) / 2 (0010) = 3 (0011) R 1 (0001)

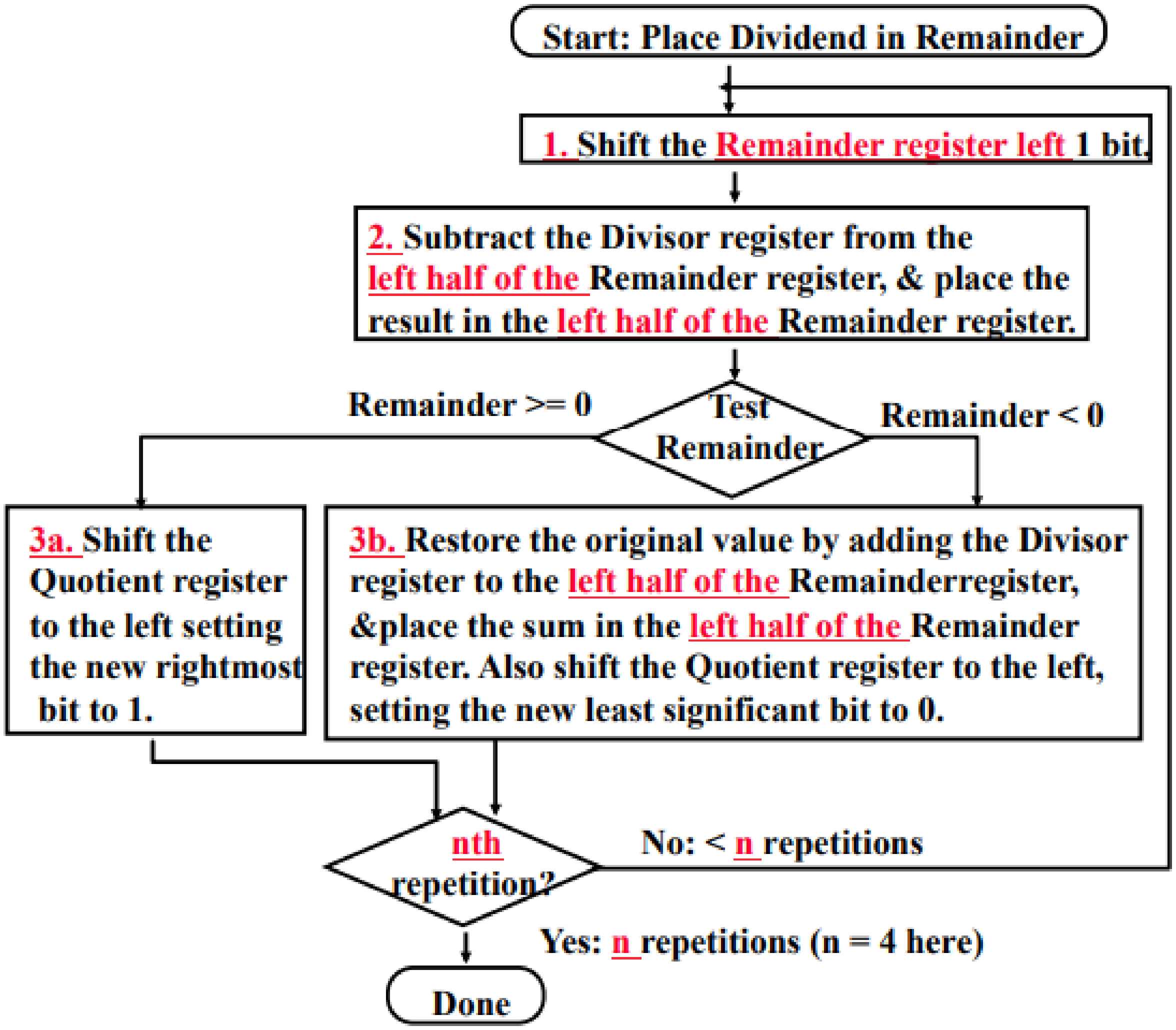
Step	Remainder	Quotient	Divisor	Rem-Div
Initial	0000 0111	0000	0010 0000	< 0
1	0000 0111	← 0000	→ 0001 0000	< 0
2	0000 0111	0000	0000 1000	< 0
3	0000 0111	0000	0000 0100	0000 0011 > 0
4	0000 0011	0001	0000 0010	0000 0001 > 0
5	0000 0001	0011	0000 0001	
Final	1	3		

- ❑ **Observations on Divide version 1:**
- ❑ Half the bits in divisor always 0
  - ⇒ 1/2 of 64-bit adder is wasted
  - ⇒ 1/2 of divisor register is wasted
- ❑ Intuition: instead of shifting divisor to right, shift remainder to left...
- ❑ Step 1 cannot produce a 1 in quotient bit – as all bits corresponding to the divisor in the remainder register are 0 (remember all operands are 32-bit)
- ❑ Intuition: switch order to shift first and then subtract – can save 1 iteration...

DIVIDE HARDWARE Version 2

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg





Start: Place Dividend in Remainder

1. Shift the Remainder register left 1 bit.

2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.

Test Remainder

Remainder >= 0

Remainder < 0

3a. Shift the Quotient register to the left setting the new rightmost bit to 1.

3b. Restore the original value by adding the Divisor register to the left half of the Remainderregister, &place the sum in the left half of the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

nth repetition?

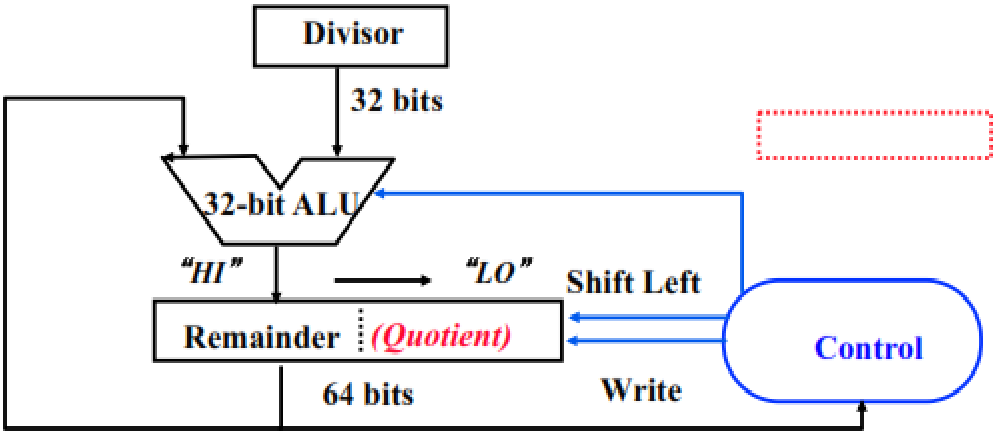
No: < n repetitions

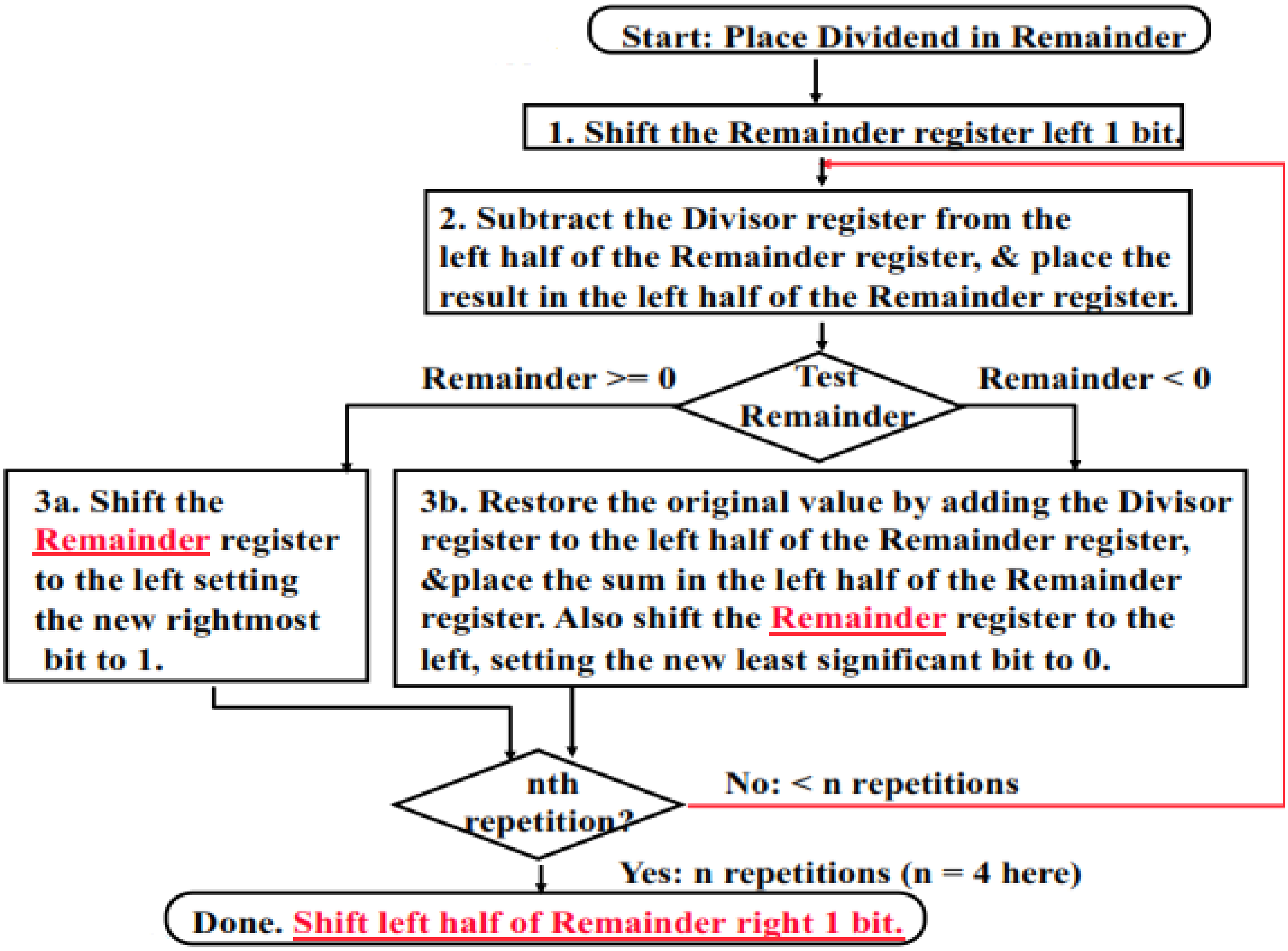
Yes: n repetitions (n = 4 here)

Done

DIVIDE HARDWARE Version 3

- 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)





Start: Place Dividend in Remainder

1. Shift the Remainder register left 1 bit.

2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.

Test Remainder

Remainder >= 0

Remainder < 0

3a. Shift the Remainder register to the left setting the new rightmost bit to 1.

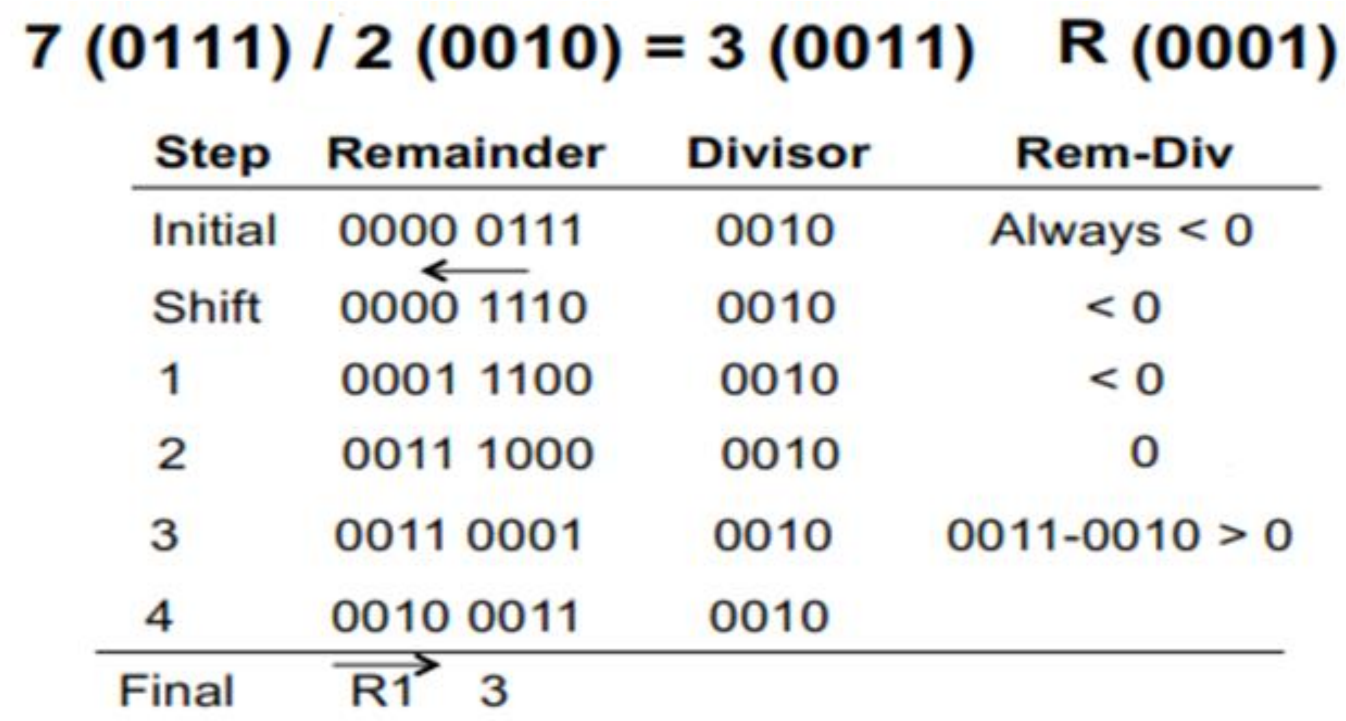
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, &place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.

nth repetition?

No: < n repetitions

Yes: n repetitions (n = 4 here)

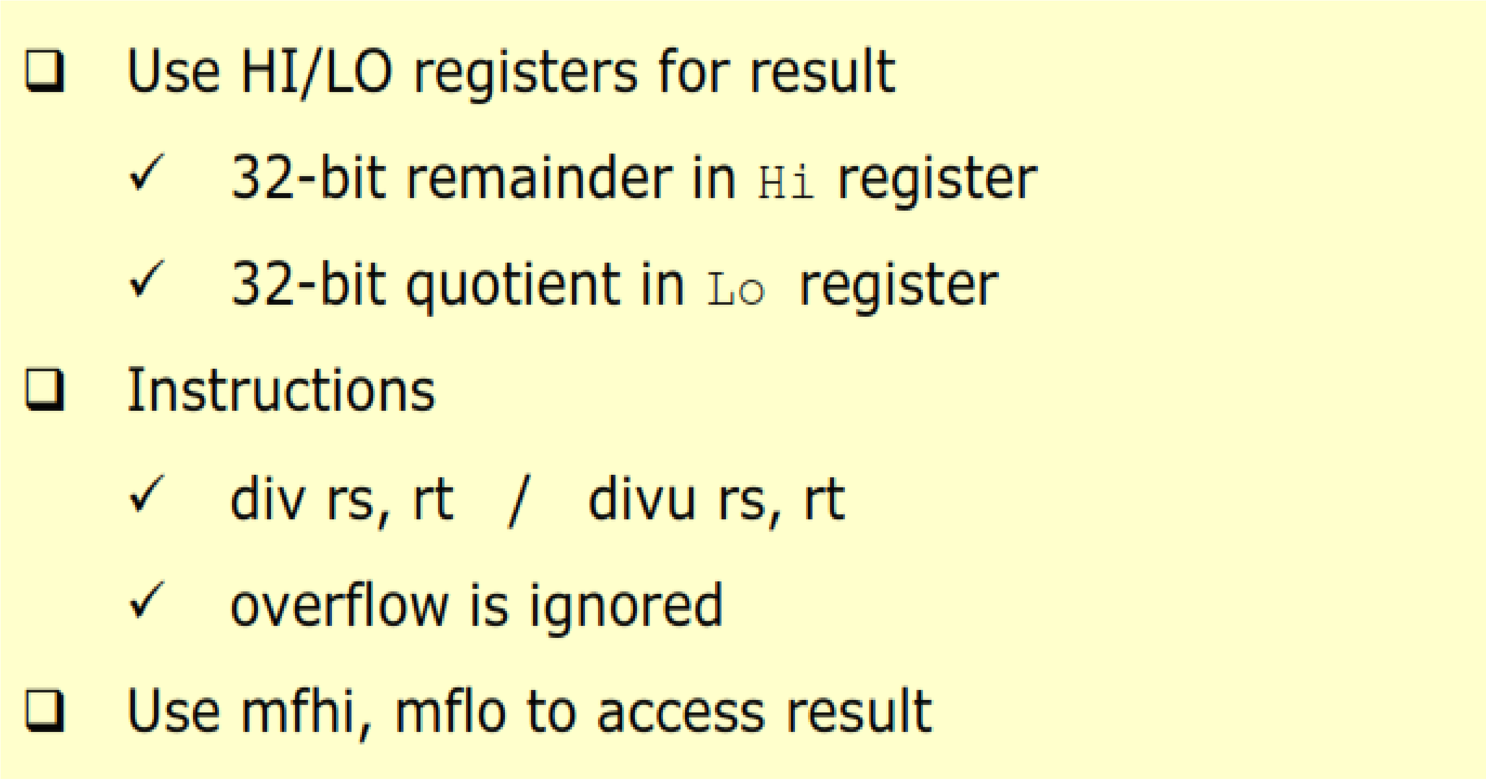
Done. Shift left half of Remainder right 1 bit.



7 (0111) / 2 (0010) = 3 (0011) R (0001)

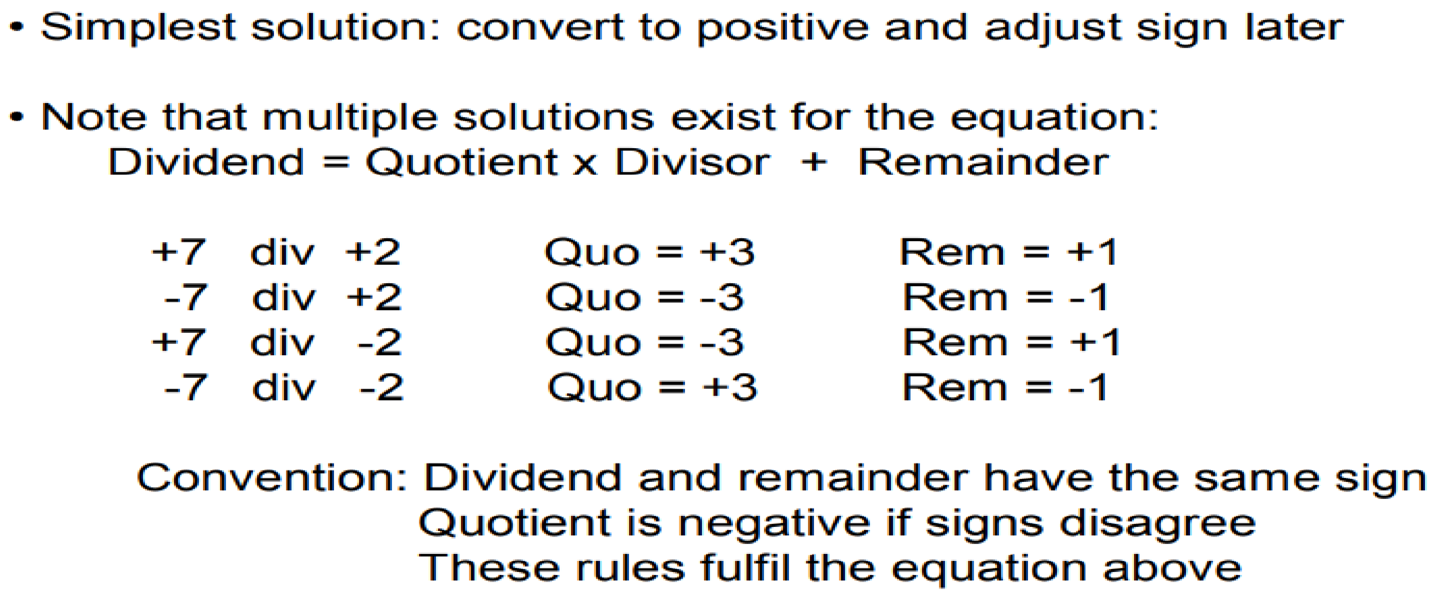
Step	Remainder	Divisor	Rem-Div
Initial	0000 0111	0010	Always < 0
Shift	0000 1110	0010	< 0
1	0001 1100	0010	< 0
2	0011 1000	0010	0
3	0011 0001	0010	0011-0010 > 0
4	0010 0011	0010	
Final	R1 3		

MIPS Division



- ❑ Use HI/LO registers for result
  - ✓ 32-bit remainder in `Hi` register
  - ✓ 32-bit quotient in `Lo` register
- ❑ Instructions
  - ✓ div rs, rt / divu rs, rt
  - ✓ overflow is ignored
- ❑ Use mfhi, mflo to access result

Divisions involving Negatives



- Simplest solution: convert to positive and adjust sign later

- Note that multiple solutions exist for the equation:
  Dividend = Quotient x Divisor + Remainder

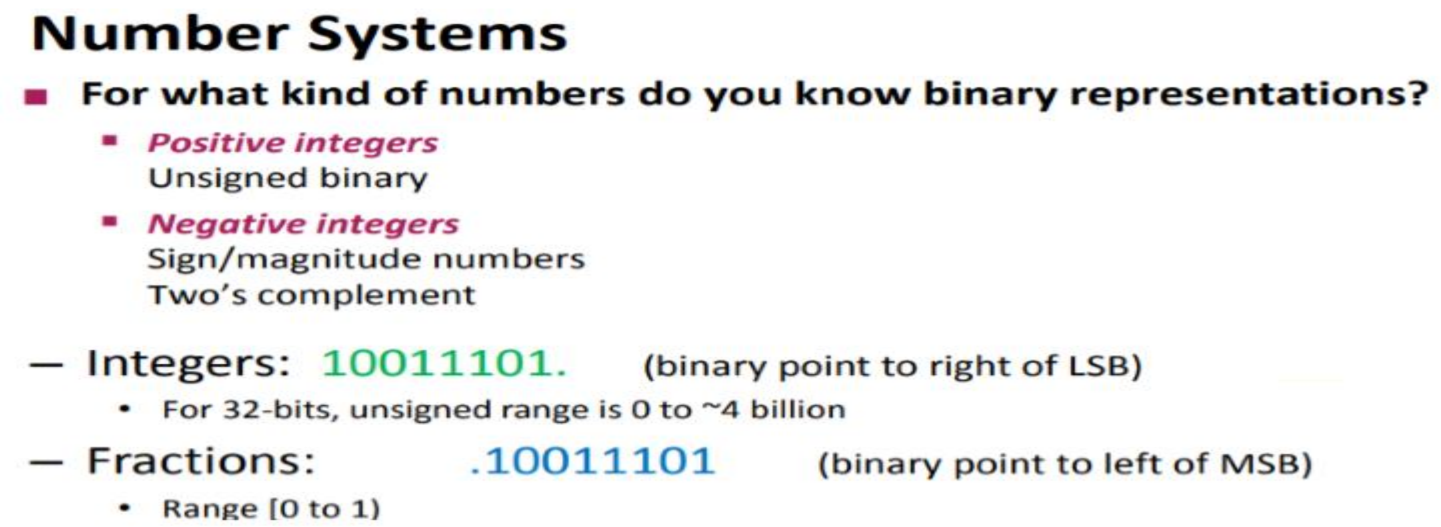
+7 div +2	Quo = +3	Rem = +1
-7 div +2	Quo = -3	Rem = -1
+7 div -2	Quo = -3	Rem = +1
-7 div -2	Quo = +3	Rem = -1

Convention: Dividend and remainder have the same sign
Quotient is negative if signs disagree
These rules fulfil the equation above

Signed Division

- Simplest way is to remember the signs
- Convert the dividend and divisor to positive
- Do the unsigned division
- Compute the signs of the quotient and remainder
  - Quotient sign = Dividend sign XOR Divisor sign
  - Remainder sign = Dividend sign

- To summarize, if dividend is negative, then two's complement must be applied to the remainder at the end. If the dividend and the divisor have different signs, then the quotient must be negated with 2's complement operation at the end.



Number Systems

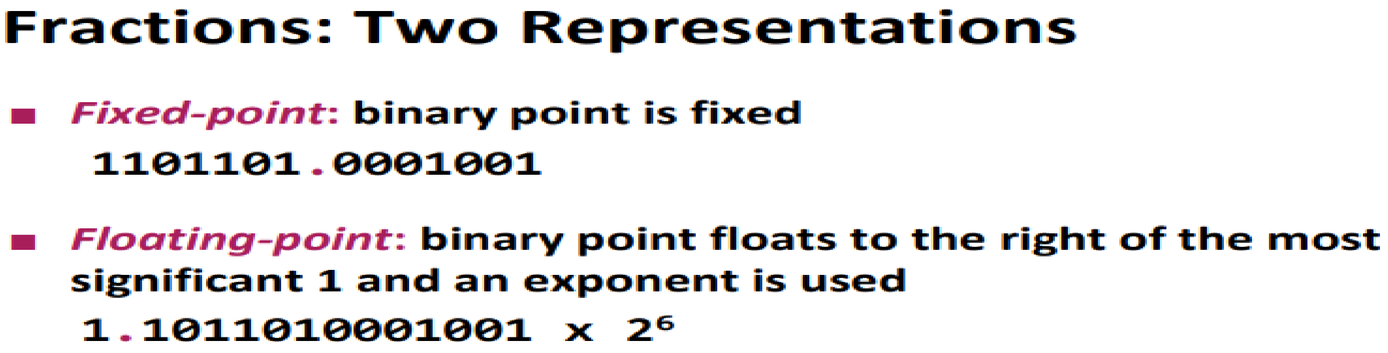
- For what kind of numbers do you know binary representations?
  - *Positive integers*
    Unsigned binary
  - *Negative integers*
    Sign/magnitude numbers
    Two's complement

— Integers: 10011101. (binary point to right of LSB)

- For 32-bits, unsigned range is 0 to ~4 billion

— Fractions: .10011101 (binary point to left of MSB)

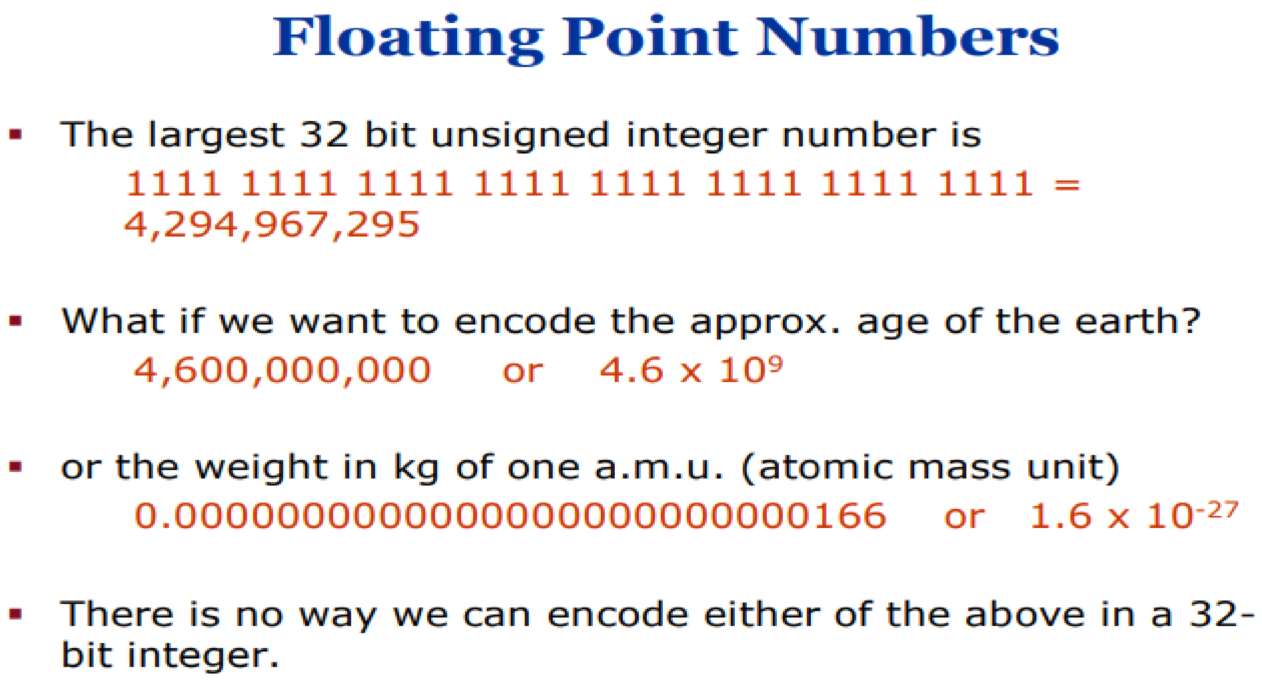
- Range [0 to 1)



Fractions: Two Representations

- *Fixed-point*: **binary point is fixed**
  1101101.0001001
- *Floating-point*: **binary point floats to the right of the most significant 1 and an exponent is used**
  1.1011010001001 x 2^6

- **Floating-point numbers have two advantages** over integers. First, they can represent values between integers. Second, because of the scaling factor, they can represent a much greater **range** of values



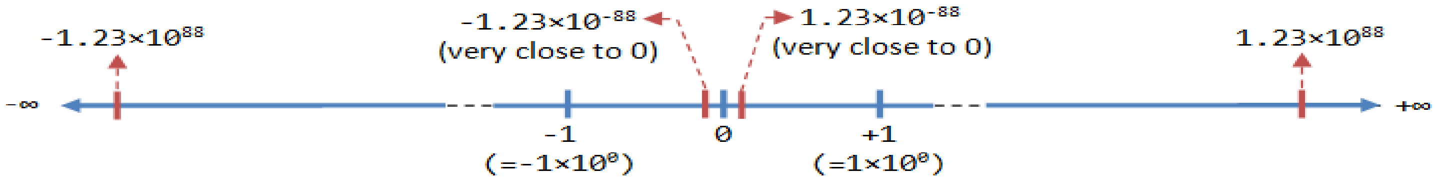
Floating Point Numbers

- The largest 32 bit unsigned integer number is
  1111 1111 1111 1111 1111 1111 1111 1111 = 4,294,967,295
- What if we want to encode the approx. age of the earth?
  4,600,000,000 or 4.6 x 10^9
- or the weight in kg of one a.m.u. (atomic mass unit)
  0.000000000000000000000000166 or 1.6 x 10^{-27}
- There is no way we can encode either of the above in a 32-bit integer.

- **The term floating point (real number)** is derived from the fact that there is no fixed number of digits before and after the radix point (Ex. decimal point); that is, the decimal point can float.

In decimal the number:	123.456	represented as	1.23456×10^2
In hexadecimal number:	123.abc	represented as	$1.23abc \times 16^2$
In binary number:	10100.110	represented as	1.0100110×2^4

- Representation for non-integral numbers.
  - Including very small and very large numbers (positive or negative)



Floating-point Numbers (Decimal)

❖We use a scientific notation to represent

- ✧Very small numbers (e.g. 1.0×10^{-9})
- ✧Very large numbers (e.g. 8.64×10^{13})
- ✧Scientific notation: $\pm\, d\,.\,fraction \times 10^{\pm\, exponent}$

- **decimal scientific notation:**
  - For example, 273_{10} in scientific notation is

    $273 = 2.73 \times 10^2$

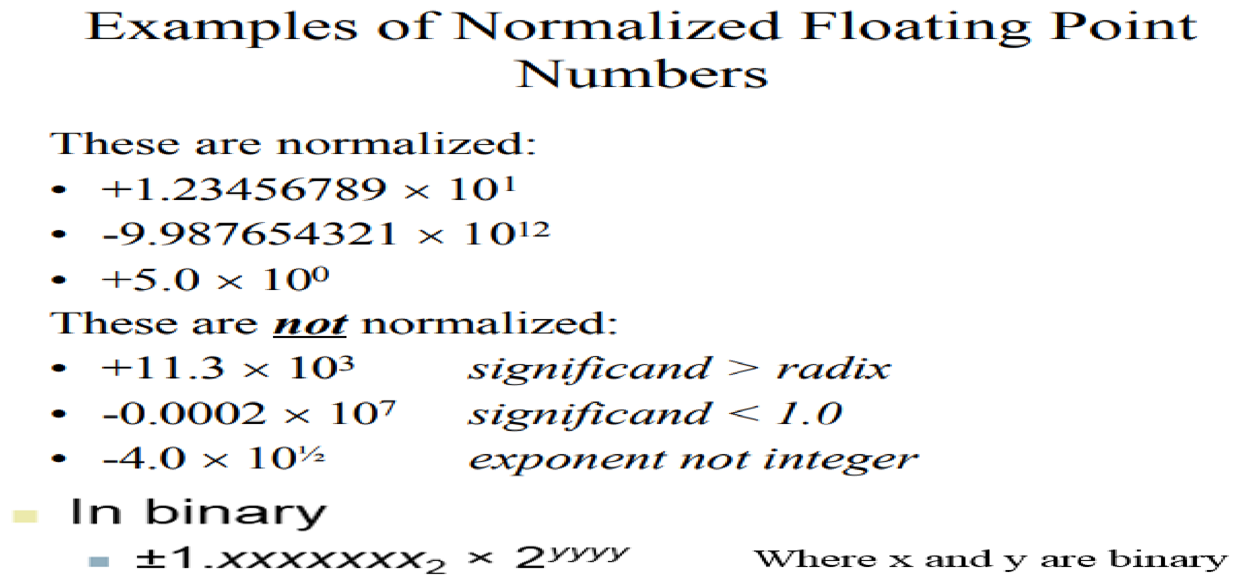
- **In general, a number is written in scientific notation as:**

  $\pm\, M \times B^E$

  Where:
  - M = mantissa
  - B = base
  - E = exponent

- **In the example, M = 2.73, B = 10, and E = 2**

- ❖ Floating-point numbers should be normalized
  - ✧ Exactly one non-zero digit should appear before the point.
    - In a decimal number, this digit can be from **1 to 9**
    - In a binary number, this digit should be **1**
  - ✧ Normalized FP Numbers:
  - ✧ 5.341×10^3



Examples of Normalized Floating Point Numbers

These are normalized:

- $+1.23456789 \times 10^1$
- $-9.987654321 \times 10^{12}$
- $+5.0 \times 10^0$

These are ***not*** normalized:

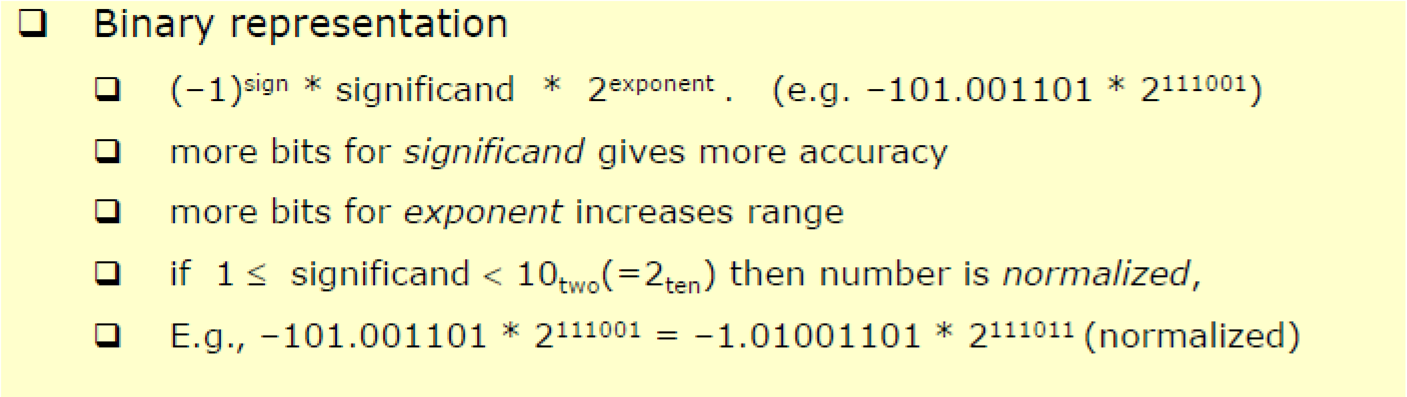
• $+11.3 \times 10^3$	*significand > radix*
• -0.0002×10^7	*significand < 1.0*
• -4.0×10^{½}	*exponent not integer*

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$ Where x and y are binary

$32_{10} = 100000_2 = 1.0 \times 2^5 = 0.1 \times 2^6$

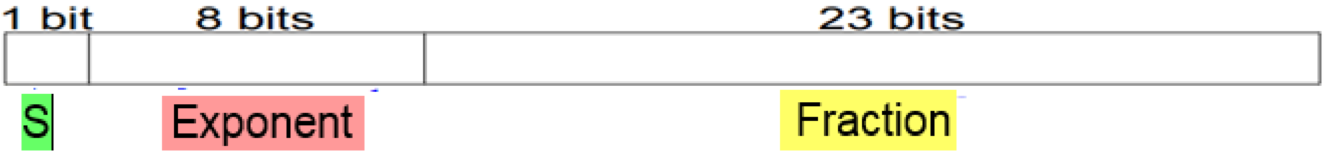
$0.0625_{10} = 0.0001_2 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$

$26.62510 = 11010.101_2 = 1.1010101 \times 2^4 = 0.11010101 \times 2^5$



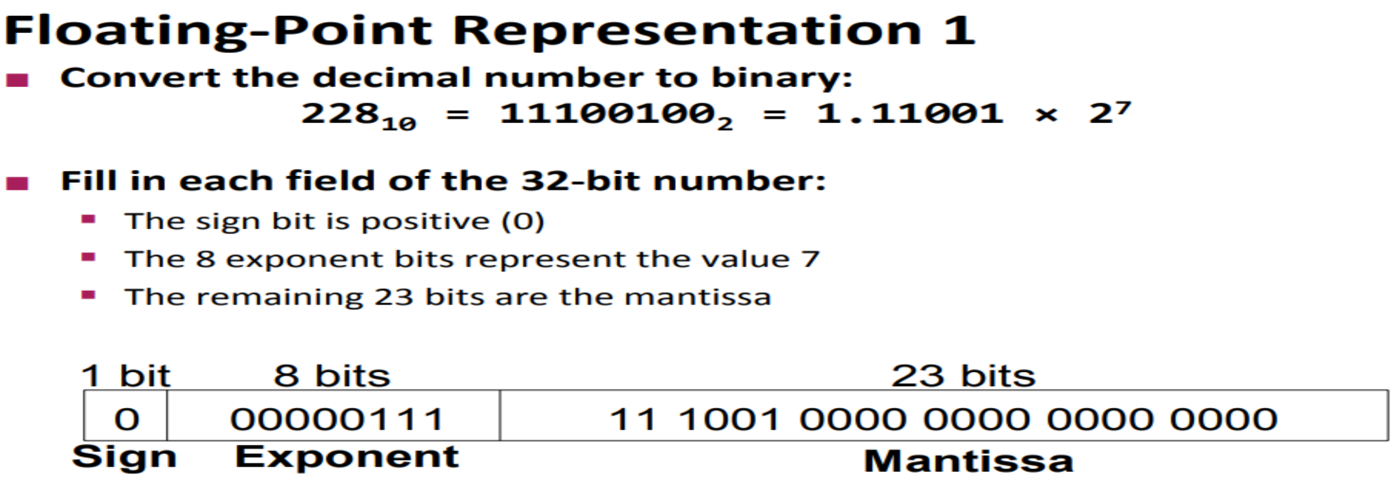
- ❑ Binary representation
  - ❑ $(-1)^{\text{sign}}$ * significand * 2^{exponent} . (e.g. $-101.001101 * 2^{111001}$)
  - ❑ more bits for *significand* gives more accuracy
  - ❑ more bits for *exponent* increases range
  - ❑ if $1 \leq$ significand $< 10_{\text{two}}(=2_{\text{ten}})$ then number is *normalized*,
  - ❑ E.g., $-101.001101 * 2^{111001} = -1.01001101 * 2^{111011}$ (normalized)

Floating-Point Representation



1 bit	8 bits	23 bits
S	Exponent	Fraction

- ✧ *S* is the Sign bit (0 is positive and 1 is negative)
- ✧ *E* is the Exponent field (signed)
  - ✧ Very large numbers have large positive exponents
  - ✧ Very small close-to-zero numbers have negative exponents
  - ✧ More bits in exponent field increases range of values
- ✧ *F* is the Fraction field (fraction after binary point)
  - ✧ More bits in fraction field improves the precision of FP numbers



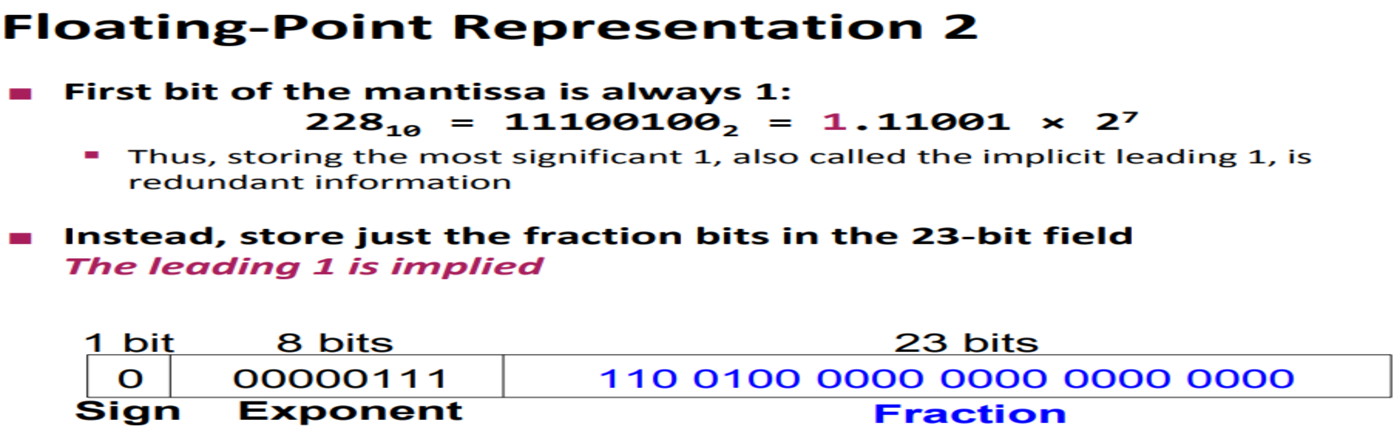
Floating-Point Representation 1

- **Convert the decimal number to binary:**

$$228_{10} = 11100100_2 = 1.11001 \times 2^7$$

- **Fill in each field of the 32-bit number:**
  - The sign bit is positive (0)
  - The 8 exponent bits represent the value 7
  - The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa**



Floating-Point Representation 2

- **First bit of the mantissa is always 1:**

$$228_{10} = 11100100_2 = 1.11001 \times 2^7$$

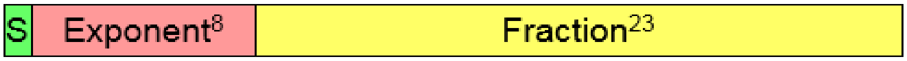
  - Thus, storing the most significant 1, also called the implicit leading 1, is redundant information

- **Instead, store just the fraction bits in the 23-bit field**
  The leading 1 is implied

1 bit	8 bits	23 bits
0	00000111	110 0100 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

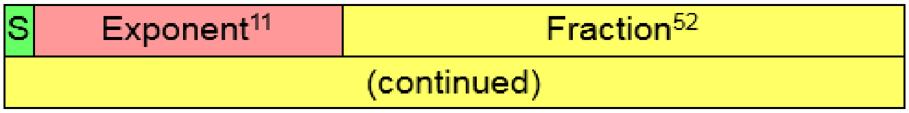
IEEE 754 Floating-Point Standard

- Found in virtually every computer invented since 1980
  - Simplified porting of floating-point numbers
  - Unified the development of floating-point algorithms
  - Increased the accuracy of floating-point numbers
- Single Precision Floating Point Numbers (32 bits)
  - 1-bit sign + 8-bit exponent + 23-bit fraction



S | Exponent^{8} | Fraction^{23}

- Double Precision Floating Point Numbers (64 bits)
  - 1-bit sign + 11-bit exponent + 52-bit fraction

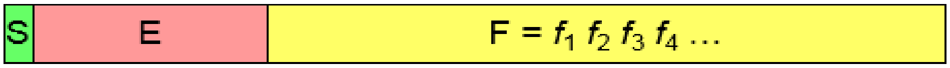


S | Exponent^{11} | Fraction^{52}

(continued)

Normalized Floating Point Numbers

- For a normalized floating point number (S, E, F)



S | E | F = $f_1\ f_2\ f_3\ f_4 \ldots$

- Significand is equal to $(1.F)_2 = (1.f_1f_2f_3f_4\ldots)_2$
  - IEEE 754 assumes hidden 1. (not stored) for normalized numbers
  - Significand is 1 bit longer than fraction
- Value of a Normalized Floating Point Number:

$$\pm (1.F)_2 \times 2^{exponent\_value}$$

$$\pm (1.f_1f_2f_3f_4 \ldots)_2 \times 2^{exponent\_value}$$

$$\pm (1 + f_1\times2^{-1} + f_2\times2^{-2} + f_3\times2^{-3} + f_4\times2^{-4} \ldots)_2 \times 2^{exponent\_value}$$

S = 0 is positive,   S = 1 is negative

Biased Exponent Representation

- How to represent a signed exponent? Choices are …
  - Sign + magnitude representation for the exponent
  - Two's complement representation
  - Biased representation
- IEEE 754 uses biased representation for the exponent
  - Exponent Value = E – Bias (Bias is a constant)
- The exponent field is 8 bits for single precision
  - E can be in the range 0 to 255
  - $E = 0$ and $E = 255$ are reserved for special use (discussed later)
  - $E = 1$ to 254 are used for normalized floating point numbers
  - Bias = 127 (half of 254)
  - Exponent value = $\mathbf{E - 127}$ Range: **-126 to +127**

- For double precision, the exponent field is 11 bits
  - E can be in the range 0 to 2047
  - $E = 0$ and $E = 2047$ are reserved for special use
  - $E = 1$ to 2046 are used for normalized floating point numbers
  - Bias = 1023 (half of 2046)
  - Exponent value = $\mathbf{E - 1023}$ Range: **-1022 to +1023**
- Value of a Normalized Floating Point Number is

$$\pm (1.F)_2 \times 2^{(E\text{-}Bias)}$$
$$\pm (1.f_1 f_2 f_3 f_4 \ldots)_2 \times 2^{(E\text{-}Bias)}$$
$$\pm (1 + f_1 \times 2^{-1} + f_2 \times 2^{-2} + f_3 \times 2^{-3} + f_4 \times 2^{-4} \ldots)_2 \times 2^{(E\text{-}Bias)}$$

S = 0 is positive, S = 1 is negative

Examples of Single Precision Float

- What is the decimal value of this Single Precision float?

1	0	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Solution:
  - Sign = 1 is negative
  - E = $(01111100)_2$ = 124, *E* – bias = 124 – 127 = –3
  - Significand = $(1.0100 \ldots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)
  - Value in decimal = -1.25×2^{-3} = –0.15625

- What is the decimal value of?

0	1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Solution:

  implicit

  - Value in decimal = $+(1.01001100 \ldots 0)_2 \times 2^{130-127}$ = $(1.01001100 \ldots 0)_2 \times 2^3 = (1010.01100 \ldots 0)_2$ = 10.375

Examples of Double Precision Float

- What is the decimal value of this Double Precision float ?

0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Solution:
  - Value of exponent = $(10000000101)_2$ – Bias = 1029 – 1023 = 6
  - Value of double = $(1.00101010 \ldots 0)_2 \times 2^6$ (1. is implicit) = $(1001010.10 \ldots 0)_2$ = 74.5

- What is the decimal value of ?

1	0	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Do it yourself! (answer should be -1.5×2^{-7} = –0.01171875)

Example

- Represent –0.75 in IEEE 754 FP
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000…00
- Double: 1011111111101000…00

Example

What number is represented by the single-precision float

1100000001000…00

S = 1

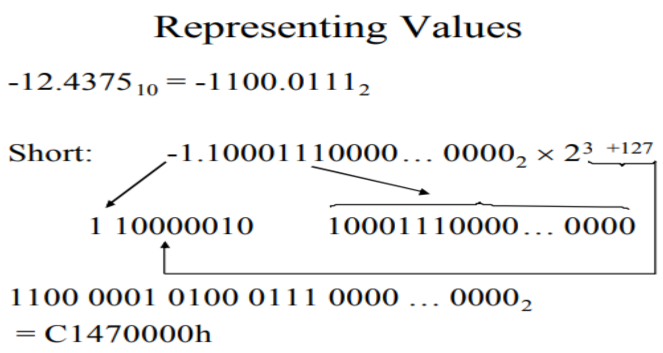
Fraction = $01000\ldots00_2$

Exponent = 10000001_2 = 129

- $x = (-1)^1 \times (1 +. 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$



Representing Values

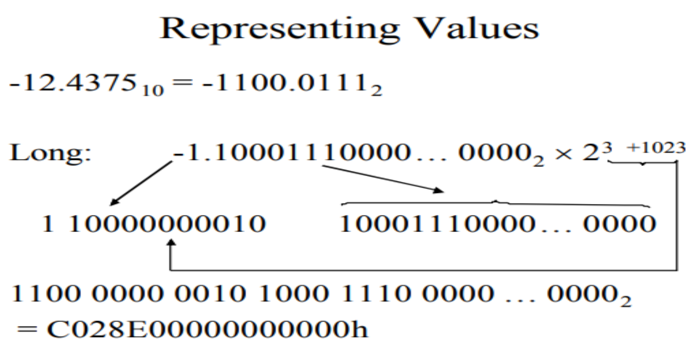
$-12.4375_{10} = -1100.0111_2$

Short: $-1.10001110000\ldots0000_2 \times 2^{3\ +127}$

1 10000010 10001110000… 0000

$1100\ 0001\ 0100\ 0111\ 0000\ \ldots\ 0000_2$

= C1470000h



Representing Values

$-12.4375_{10} = -1100.0111_2$

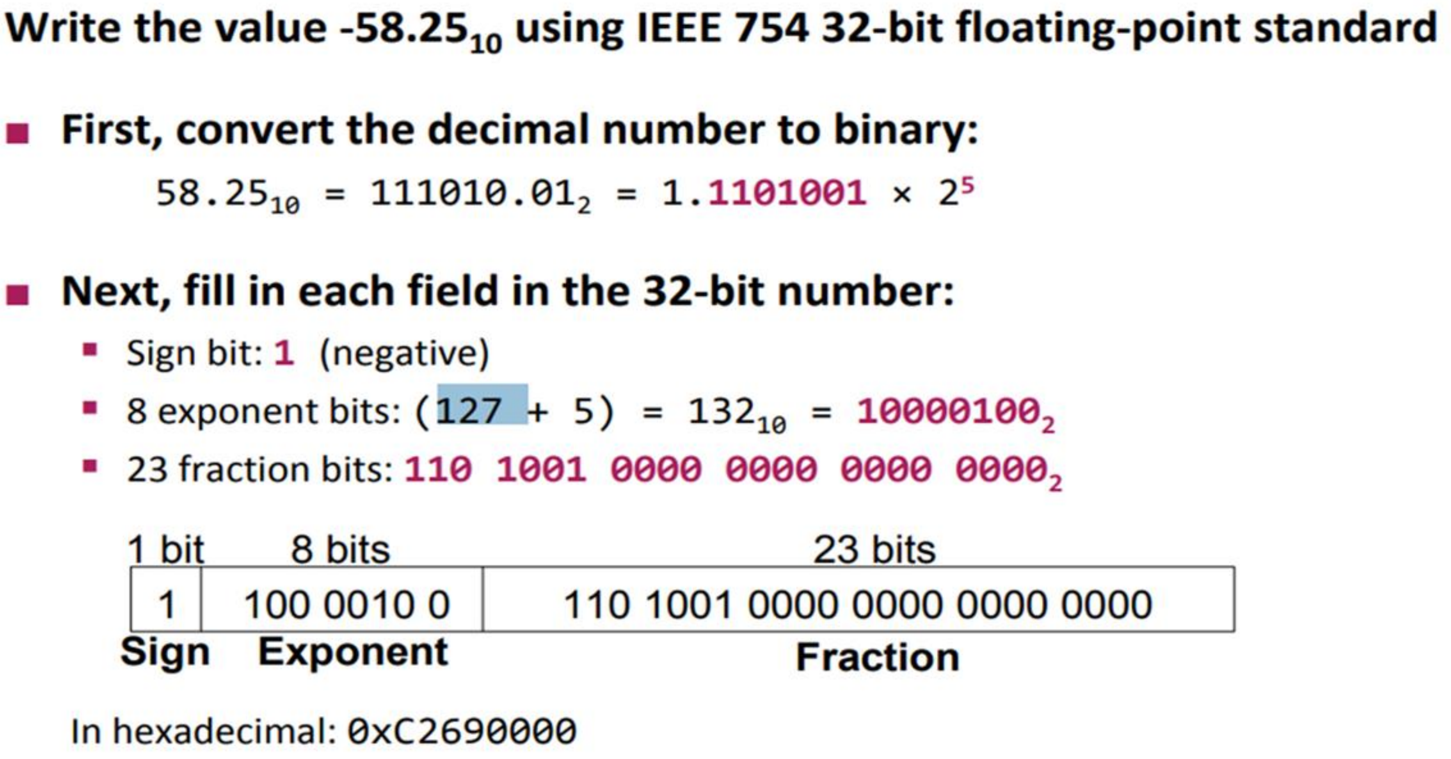
Long: $-1.10001110000\ldots0000_2 \times 2^{3\ +1023}$

1 10000000010 10001110000… 0000

$1100\ 0000\ 0010\ 1000\ 1110\ 0000\ \ldots\ 0000_2$

= C028E00000000000h

EX



Write the value -58.25_{10} using IEEE 754 32-bit floating-point standard

- First, convert the decimal number to binary:

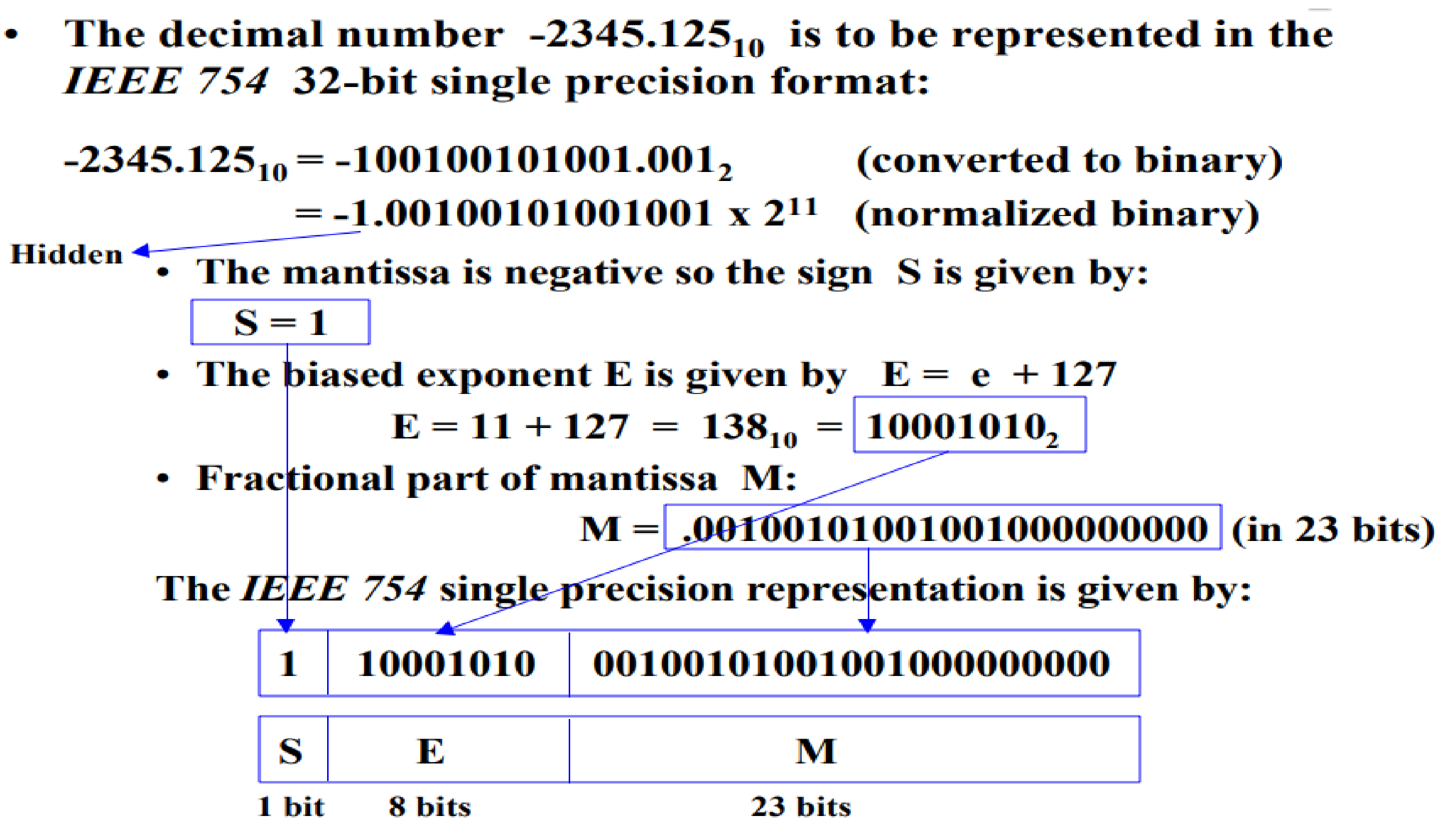
  $58.25_{10} = 111010.01_2 = 1.1101001 \times 2^5$

- Next, fill in each field in the 32-bit number:
  - Sign bit: 1 (negative)
  - 8 exponent bits: $(127 + 5) = 132_{10} = 10000100_2$
  - 23 fraction bits: $110\ 1001\ 0000\ 0000\ 0000\ 0000_2$

1 bit	8 bits	23 bits
1	100 0010 0	110 1001 0000 0000 0000 0000
Sign	Exponent	Fraction

In hexadecimal: 0xC2690000

Example



- The decimal number -2345.125_{10} is to be represented in the *IEEE 754* 32-bit single precision format:

$$-2345.125_{10} = -100100101001.001_2 \quad \text{(converted to binary)}$$

$$= -1.00100101001001 \times 2^{11} \quad \text{(normalized binary)}$$

Hidden

- The mantissa is negative so the sign S is given by:

  $\boxed{S = 1}$

- The biased exponent E is given by E = e + 127

  $E = 11 + 127 = 138_{10} = \boxed{10001010_2}$

- Fractional part of mantissa M:

  $M = \boxed{.00100101001001000000000}$ (in 23 bits)

The *IEEE 754* single precision representation is given by:

1	10001010	00100101001001000000000
S	E	M
1 bit	8 bits	23 bits

Smallest Normalized Float

- What is the smallest (in absolute value) normalized float?
- Solution for Single Precision:

0 00000001 00000000000000000000000

  - Exponent – bias = 1 – 127 = -126 (smallest exponent for SP)
  - Significand = $(1.000 \dots 0)_2 = 1$
  - Value in decimal = 1 × 2^{-126} = 1.17549 … × 10^{-38}
- Solution for Double Precision:

0 00000000001 00000000000000000000

00000000000000000000000000000000

  - Value in decimal = 1 × 2^{-1022} = 2.22507 … × 10^{-308}
- Underflow: exponent is too small to fit in exponent field

Largest Normalized Float

- What is the Largest normalized float?
- Solution for Single Precision:

0 11111110 11111111111111111111111

  - E – bias = 254 – 127 = +127 (largest exponent for SP)
  - Significand = $(1.111 \dots 1)_2$ = 1.99999988 = almost 2
  - Value in decimal ≈ 2 × 2^{+127} ≈ 2^{+128} ≈ 3.4028 … × 10^{+38}
- Solution for Double Precision:

0 11111111110 11111111111111111111

11111111111111111111111111111111

  - Value in decimal ≈ 2 × 2^{+1023} ≈ 2^{+1024} ≈ 1.79769 … × 10^{+308}
- Overflow: exponent is too large to fit in the exponent field

Zero, Infinity, and NaN

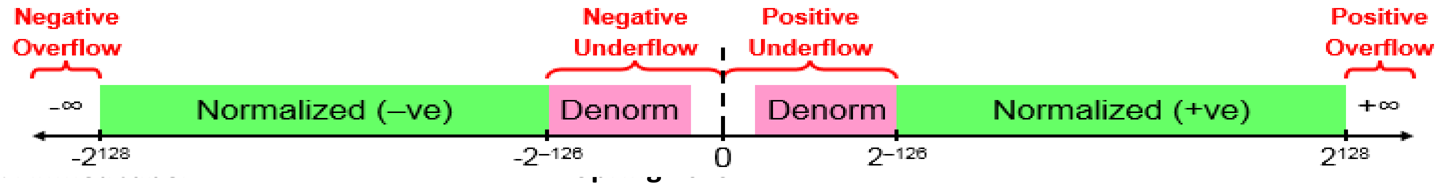
- Zero
  - Exponent field $E = 0$ and fraction $F = 0$
  - **+0** and **-0** are both possible according to sign bit S
- Infinity
  - Infinity is a special value represented with maximum E and $F = 0$
    - For single precision with 8-bit exponent: maximum $E = 255$
    - For double precision with 11-bit exponent: maximum $E = 2047$
  - Infinity can result from overflow or division by zero
  - **+∞** and **-∞** are both possible according to sign bit S
- NaN (Not a Number)
  - NaN is a special value represented with maximum E and $F \neq 0$
  - **0 / 0 → NaN, 0 × ∞ → NaN, sqrt(-1) → NaN**
  - Operation on a NaN is typically a NaN: **Op(*X*, NaN) → NaN**

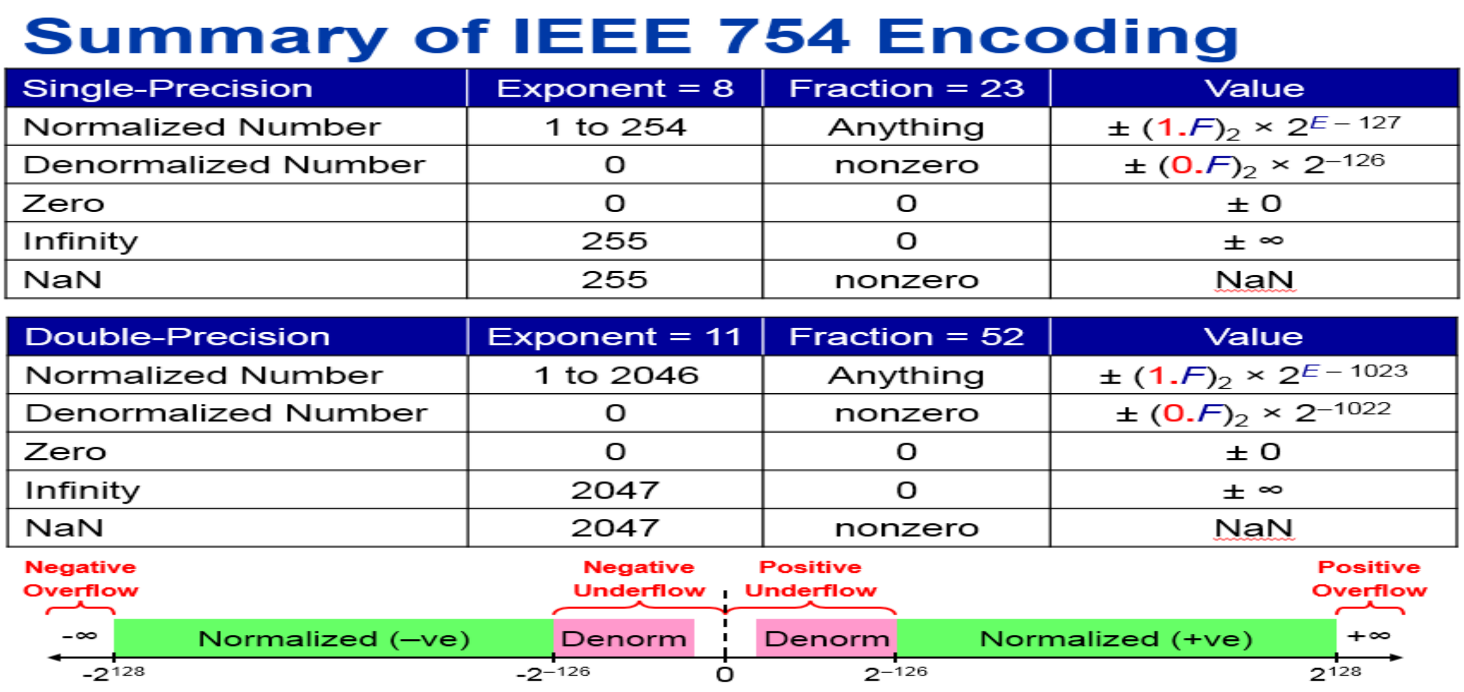
Denormalized Numbers

- IEEE standard uses denormalized numbers to …
  - Fill the gap between 0 and the smallest normalized float
  - Provide gradual underflow to zero
- Denormalized: exponent field E is 0 and fraction $F \neq 0$
  - The Implicit 1. before the fraction now becomes 0. (denormalized)
- Value of denormalized number (S, 0, F)

Single precision: $\pm (0.F)_2 \times 2^{-126}$

Double precision: $\pm (0.F)_2 \times 2^{-1022}$



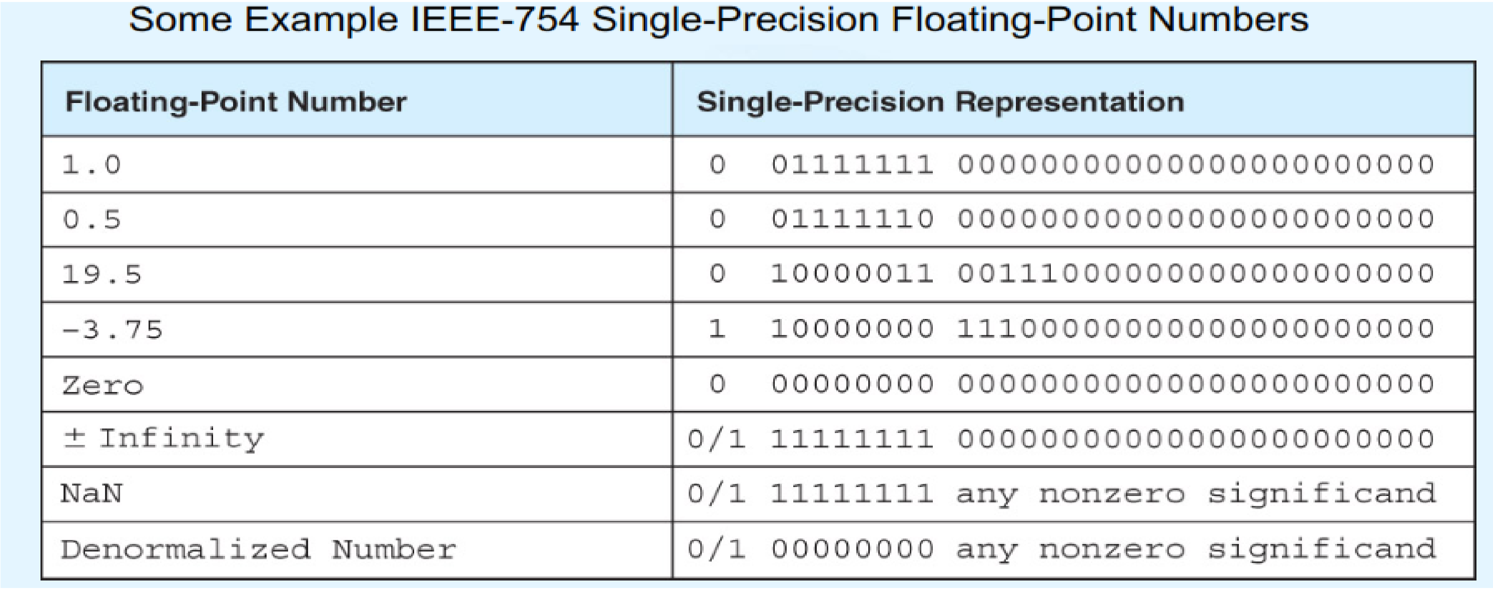


Summary of IEEE 754 Encoding

Single-Precision	Exponent = 8	Fraction = 23	Value
Normalized Number	1 to 254	Anything	$\pm (1.F)_2 \times 2^{E-127}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-126}$
Zero	0	0	± 0
Infinity	255	0	± ∞
NaN	255	nonzero	NaN

Double-Precision	Exponent = 11	Fraction = 52	Value
Normalized Number	1 to 2046	Anything	$\pm (1.F)_2 \times 2^{E-1023}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-1022}$
Zero	0	0	± 0
Infinity	2047	0	± ∞
NaN	2047	nonzero	NaN

Negative Overflow
Negative Underflow
Positive Underflow
Positive Overflow
-∞
Normalized (−ve)
Denorm
Denorm
Normalized (+ve)
+∞
-2^{128}
-2^{-126}
0
2^{-126}
2^{128}



Some Example IEEE-754 Single-Precision Floating-Point Numbers

Floating-Point Number	Single-Precision Representation
1.0	0 01111111 00000000000000000000000
0.5	0 01111110 00000000000000000000000
19.5	0 10000011 00111000000000000000000
-3.75	1 10000000 11100000000000000000000
Zero	0 00000000 00000000000000000000000
± Infinity	0/1 11111111 00000000000000000000000
NaN	0/1 11111111 any nonzero significand
Denormalized Number	0/1 00000000 any nonzero significand

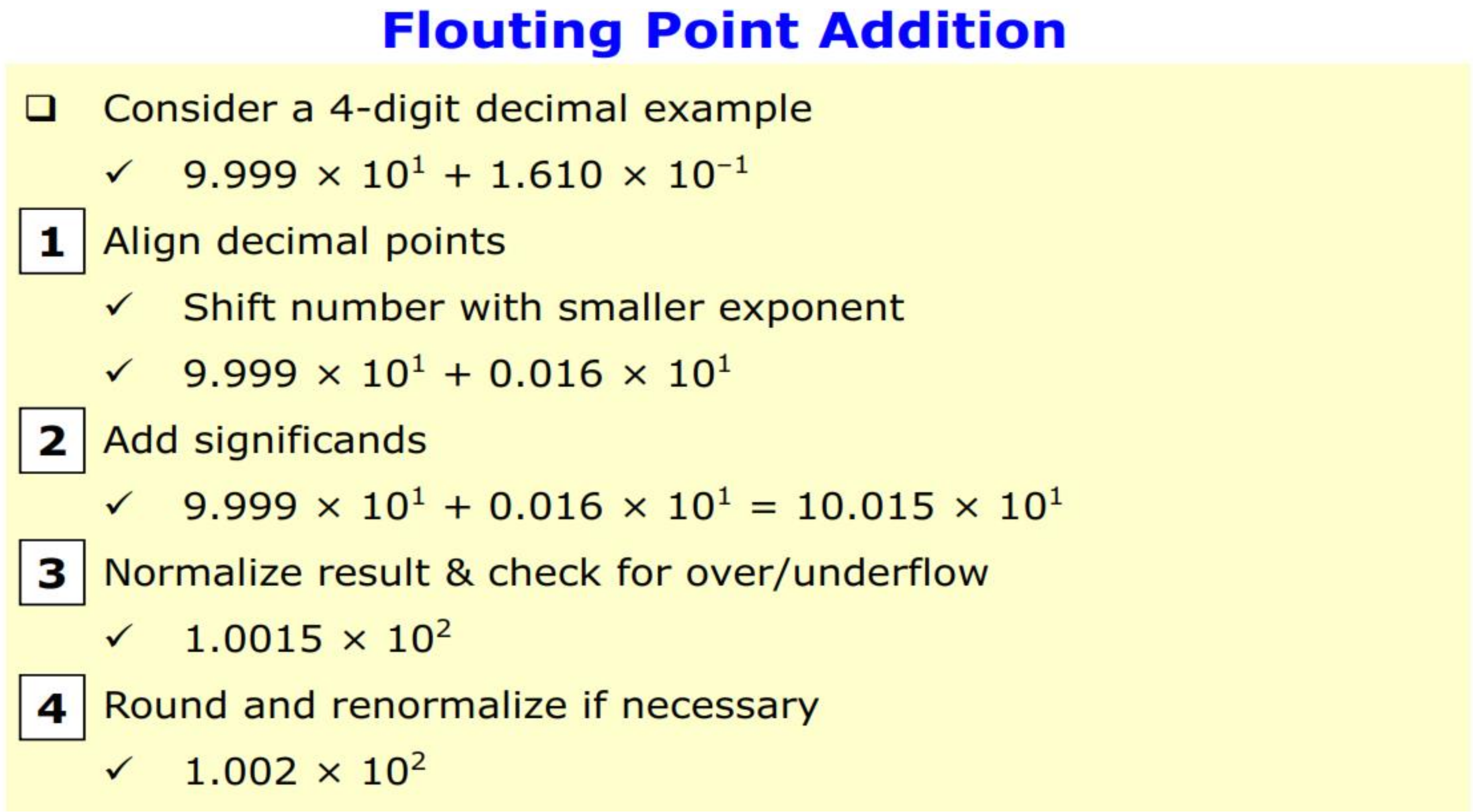
If the real exponent of a number is X then it is represented as (X + bias). IEEE single-precision uses a bias of **127**. Therefore, an exponent of

-1 is represented as -1 + 127 = 126 = 01111110_2

0 is represented as 0 + 127 = 127 = 01111111_2

+1 is represented as +1 + 127 = 128 = 10000000_2

+5 is represented as +5 + 127 = 132 = 10000100_2



Flouting Point Addition

❑ Consider a 4-digit decimal example

✓ $9.999 \times 10^{1} + 1.610 \times 10^{-1}$

1 Align decimal points

✓ Shift number with smaller exponent

✓ $9.999 \times 10^{1} + 0.016 \times 10^{1}$

2 Add significands

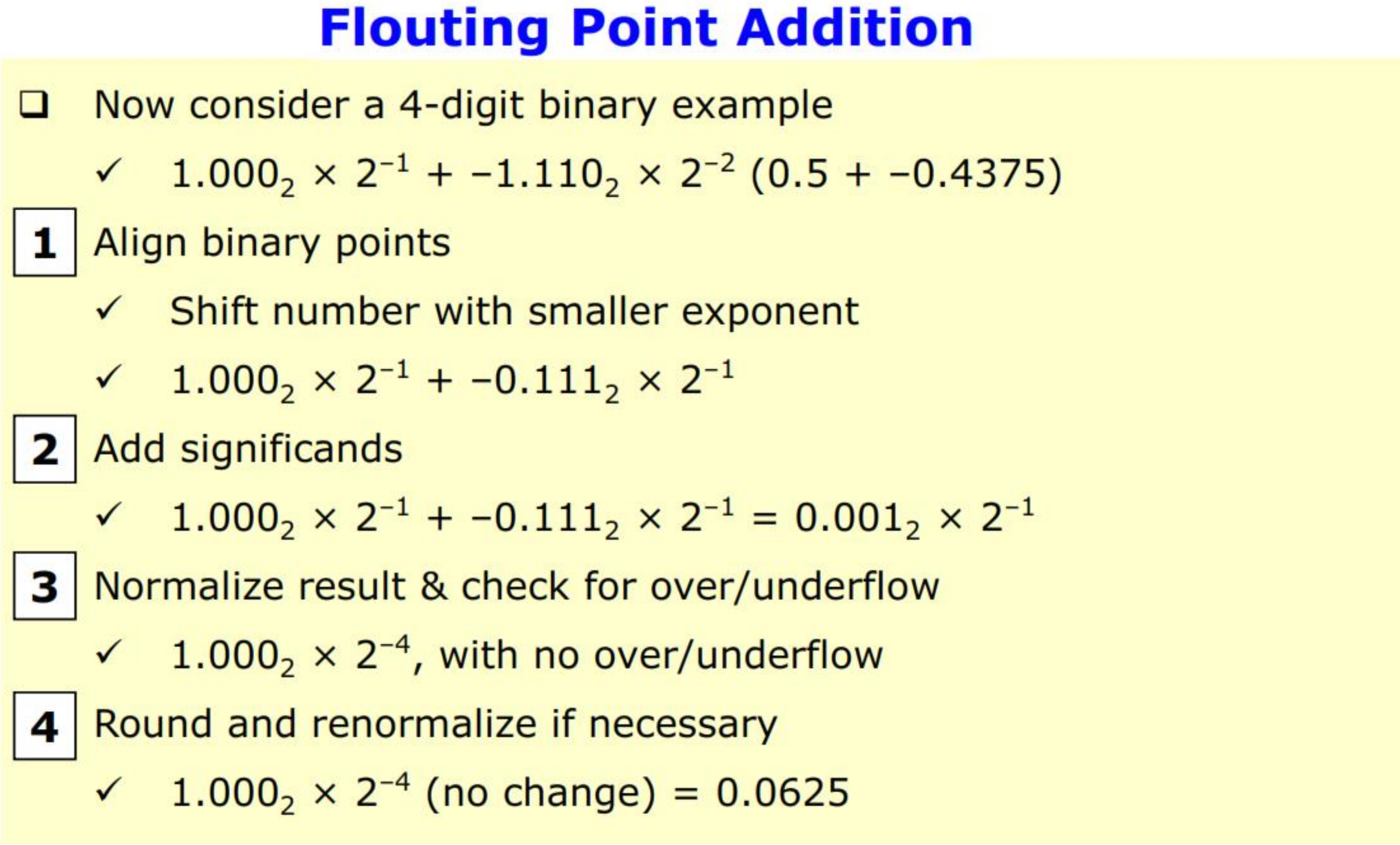
✓ $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$

3 Normalize result & check for over/underflow

✓ 1.0015×10^{2}

4 Round and renormalize if necessary

✓ 1.002×10^{2}



Flouting Point Addition

❑ Now consider a 4-digit binary example

✓ $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + −0.4375)

1 Align binary points

✓ Shift number with smaller exponent

✓ $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

2 Add significands

✓ $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

3 Normalize result & check for over/underflow

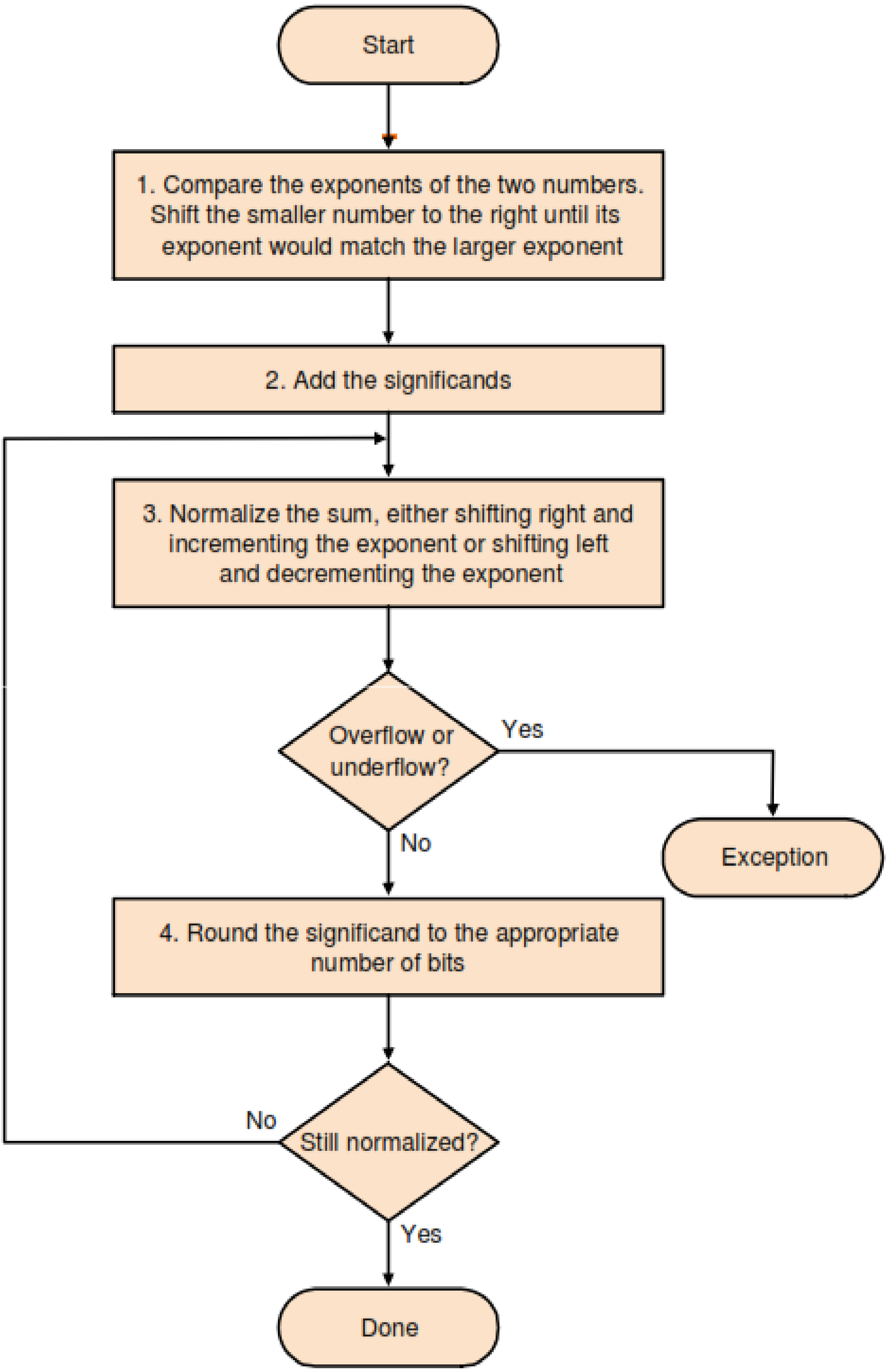
✓ $1.000_2 \times 2^{-4}$, with no over/underflow

4 Round and renormalize if necessary

✓ $1.000_2 \times 2^{-4}$ (no change) = 0.0625

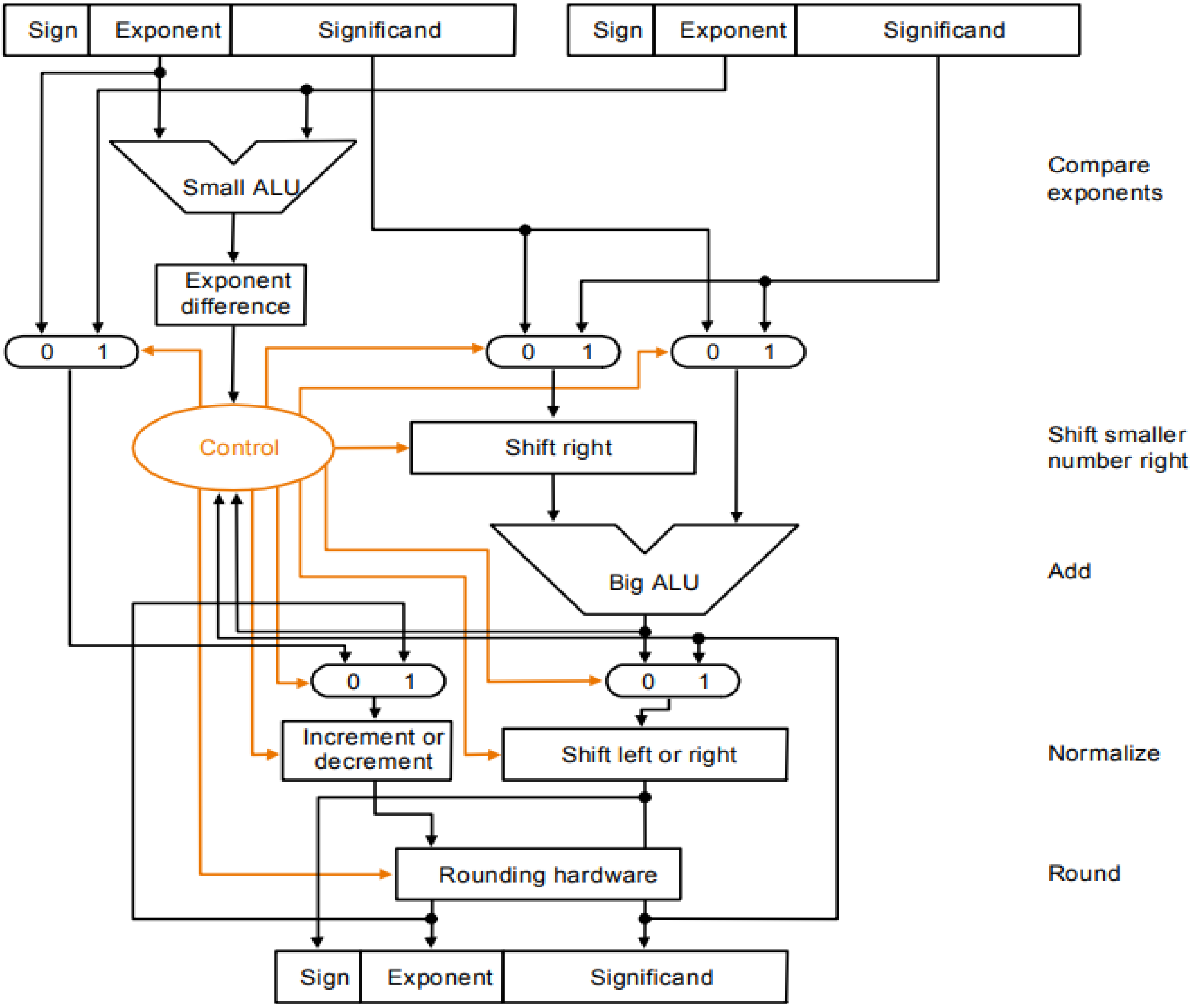
FP Adder

❑ **Algorithm**


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined



Example

- Consider adding: $(1.111)_2 \times 2^{-1} + (1.011)_2 \times 2^{-3}$
  - For simplicity, we assume 4 bits of precision (or 3 bits of fraction)
- Cannot add significands … Why?
  - Because exponents are not equal
- How to make exponents equal?
  - Shift the significand of the lesser exponent right until its exponent matches the larger number
- $(1.011)_2 \times 2^{-3} = (0.1011)_2 \times 2^{-2} = (0.01011)_2 \times 2^{-1}$
  - Difference between the two exponents = $-1 - (-3) = 2$
  - So, shift right by 2 bits
- Now, add the significands:

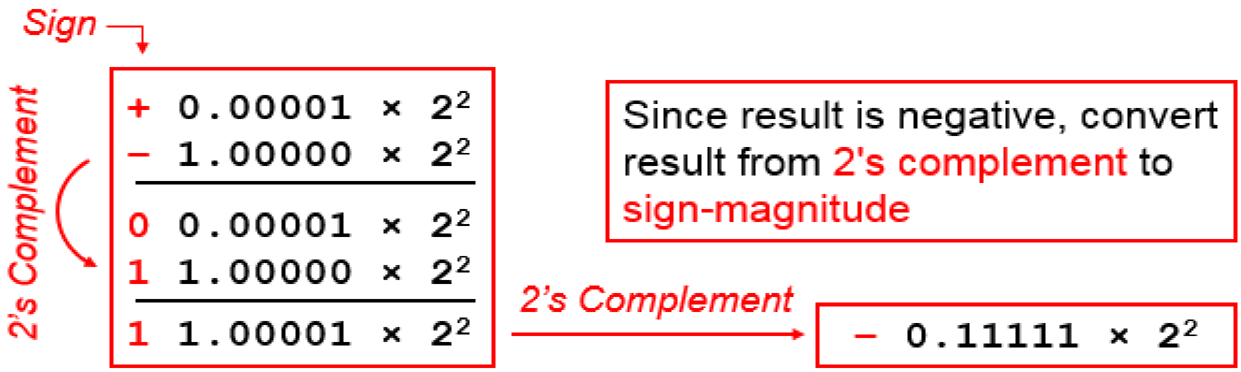
$$\begin{array}{lrr} & & 1.111 \\ & + & 0.01011 \\ \hline \textit{Carry} \rightarrow & & 10.00111 \end{array}$$

- So, $(1.111)_2 \times 2^{-1} + (1.011)_2 \times 2^{-3} = (10.00111)_2 \times 2^{-1}$
- However, result $(10.00111)_2 \times 2^{-1}$ is NOT normalized
- Normalize result: $(10.00111)_2 \times 2^{-1} = (1.000111)_2 \times 2^{0}$
  - In this example, we have a carry
  - So, shift right by 1 bit and increment the exponent
- Round the significand to fit in appropriate number of bits
  - We assumed 4 bits of precision or 3 bits of fraction
- Round to nearest: $(1.000111)_2 \approx (1.001)_2$
  - Renormalize if rounding generates a carry
- Detect overflow / underflow
  - If exponent becomes too large (overflow) or too small (underflow)

$$\begin{array}{lr|l} & 1.000 & 111 \\ + & 1 & \hookleftarrow \\ \hline & 1.001 & \end{array}$$

Example

- Consider: $(1.000)_2 \times 2^{-3} - (1.000)_2 \times 2^{2}$
  - We assume again: 4 bits of precision (or 3 bits of fraction)
- Shift significand of the lesser exponent right
  - Difference between the two exponents = 2 – (–3) = 5
  - Shift right by 5 bits: $(1.000)_2 \times 2^{-3} = (0.00001000)_2 \times 2^{2}$
- Convert subtraction into addition to 2's complement



Sign

2's Complement

```
+ 0.00001 × 2²
– 1.00000 × 2²
0 0.00001 × 2²
1 1.00000 × 2²
1 1.00001 × 2²
```

Since result is negative, convert result from 2's complement to sign-magnitude

2's Complement

$- 0.11111 \times 2^{2}$

- So, $(1.000)_2 \times 2^{-3} - (1.000)_2 \times 2^{2} = - 0.11111_2 \times 2^{2}$
- Normalize result: $- 0.11111_2 \times 2^{2} = - 1.1111_2 \times 2^{1}$
  - For subtraction, we can have leading zeros
  - Count number *z* of leading zeros (in this case *z* = 1)
  - Shift left and decrement exponent by *z*
- Round the significand to fit in appropriate number of bits
  - We assumed 4 bits of precision or 3 bits of fraction
- Round to nearest: $(1.1111)_2 \approx (10.000)_2$

```
   1.111 1
+        1
  10.000
```

- Renormalize: rounding generated a carry

  $-1.1111_2 \times 2^{1} \approx -10.000_2 \times 2^{1} = -1.000_2 \times 2^{2}$
  - Result would have been accurate if more fraction bits are used

Example

- Consider Adding Single-Precision Floats:

$$\begin{array}{rl} & 1.11100100000000000000010_2 \times 2^{4} \\ + & 1.10000000000000110000101_2 \times 2^{2} \end{array}$$

- Cannot add significands … Why?
  - Because exponents are not equal
- How to make exponents equal?
  - Shift the significand of the lesser exponent right
  - Difference between the two exponents = 4 – 2 = 2
  - So, shift right second number by 2 bits and increment exponent

$$\begin{array}{rll} & 1.10000000000000110000101_2 & \times 2^{2} \\ = & 0.01100000000000001100001\ 01_2 & \times 2^{4} \end{array}$$

- Now, ADD the Significands:

$$\begin{array}{rlll} & 1.11100100000000000000010 & \times 2^{4} & \\ + & 1.10000000000000110000101 & \times 2^{2} & \\ \hline & 1.11100100000000000000010 & \times 2^{4} & \\ + & 0.01100000000000001100001\ 01 & \times 2^{4} & \text{(shift right)} \\ \hline & 10.01000100000000001100011\ 01 & \times 2^{4} & \text{(result)} \end{array}$$

- Addition produces a carry bit, result is NOT normalized
- Normalize Result (shift right and increment exponent):

$$\begin{array}{rlll} & 10.01000100000000001100011\ 01 & \times 2^{4} & \\ \hline = & 1.00100010000000000110001\ 101 & \times 2^{5} & \text{(normalized)} \end{array}$$

Rounding

- Single-precision requires only 23 fraction bits
- However, Normalized result can contain additional bits

```
1.00100010000000000110001 | 1 01 × 2^5
```

Round Bit: **R = 1** *Sticky Bit:* **S = 1**

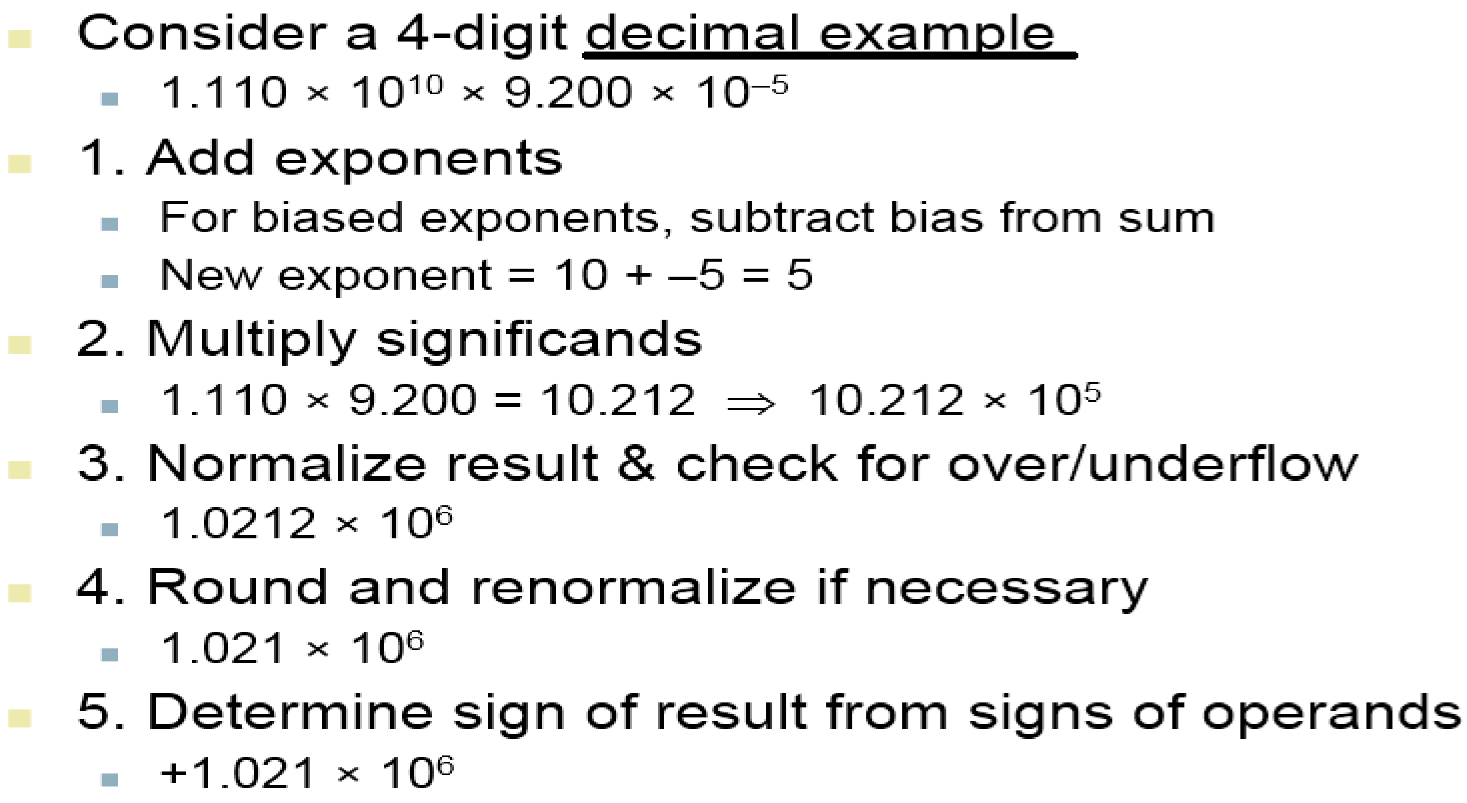
- Two extra bits are used for rounding
  - **Round bit:** appears just after the normalized result
  - **Sticky bit:** appears after the round bit (OR of all additional bits)
- Since **RS = 11**, increment fraction to **round to nearest**

```
1.00100010000000000110001 × 2^5
                       +1
-------------------------------
1.00100010000000000110010 × 2^5 (Rounded)
```

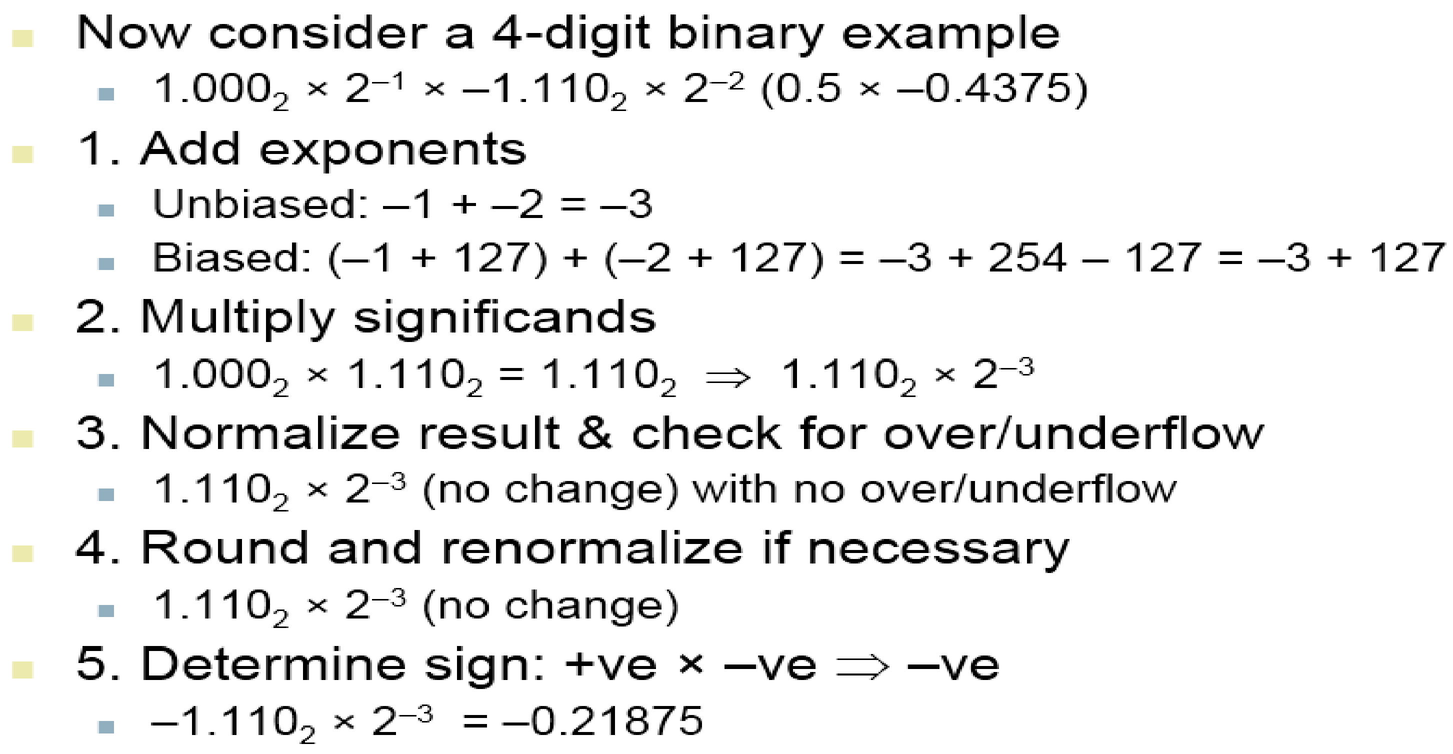
Rounding to Nearest Even

- Normalized result has the form: **1.** $f_1\ f_2\ \ldots\ f_l$ **R S**
  - The round bit **R** appears immediately after the last fraction bit f_l
  - The sticky bit **S** is the OR of all remaining additional bits
- Round to Nearest Even: default rounding mode
- Four cases for **RS**:
  - **RS = 00** → Result is Exact, no need for rounding
  - **RS = 01** → **Truncate** result by discarding **RS**
  - **RS = 11** → **Increment** result: ADD 1 to last fraction bit
  - **RS = 10** → Tie Case (either truncate or increment result)
    - **Check Last fraction bit** f_l (f_{23} for single-precision or f_{52} for double)
    - **If f_l is 0 then truncate result to keep fraction even**
    - **If f_l is 1 then increment result to make fraction even**

Floating-Point Multiplication



- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$



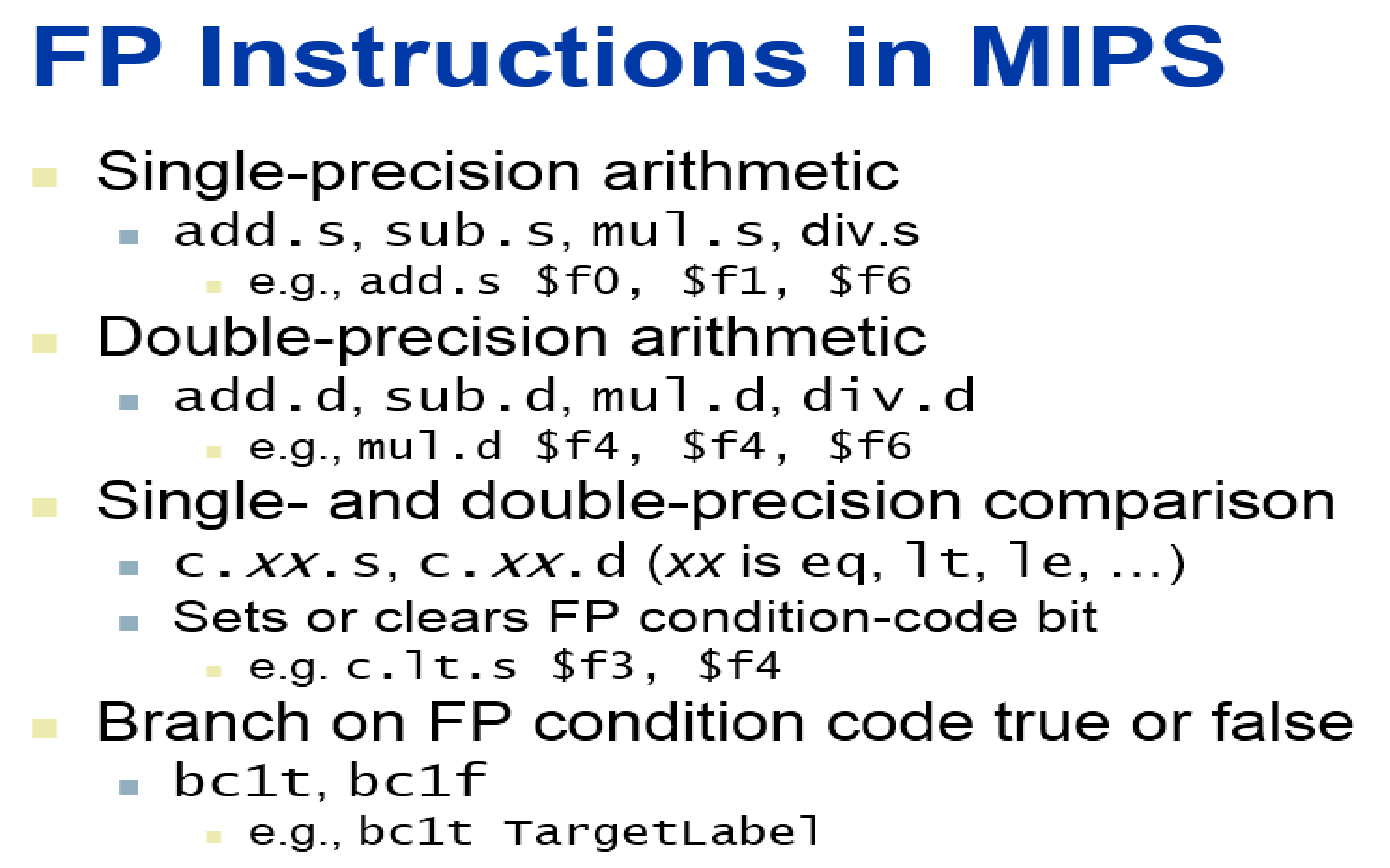
- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
  - Unbiased: $-1 + -2 = -3$
  - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

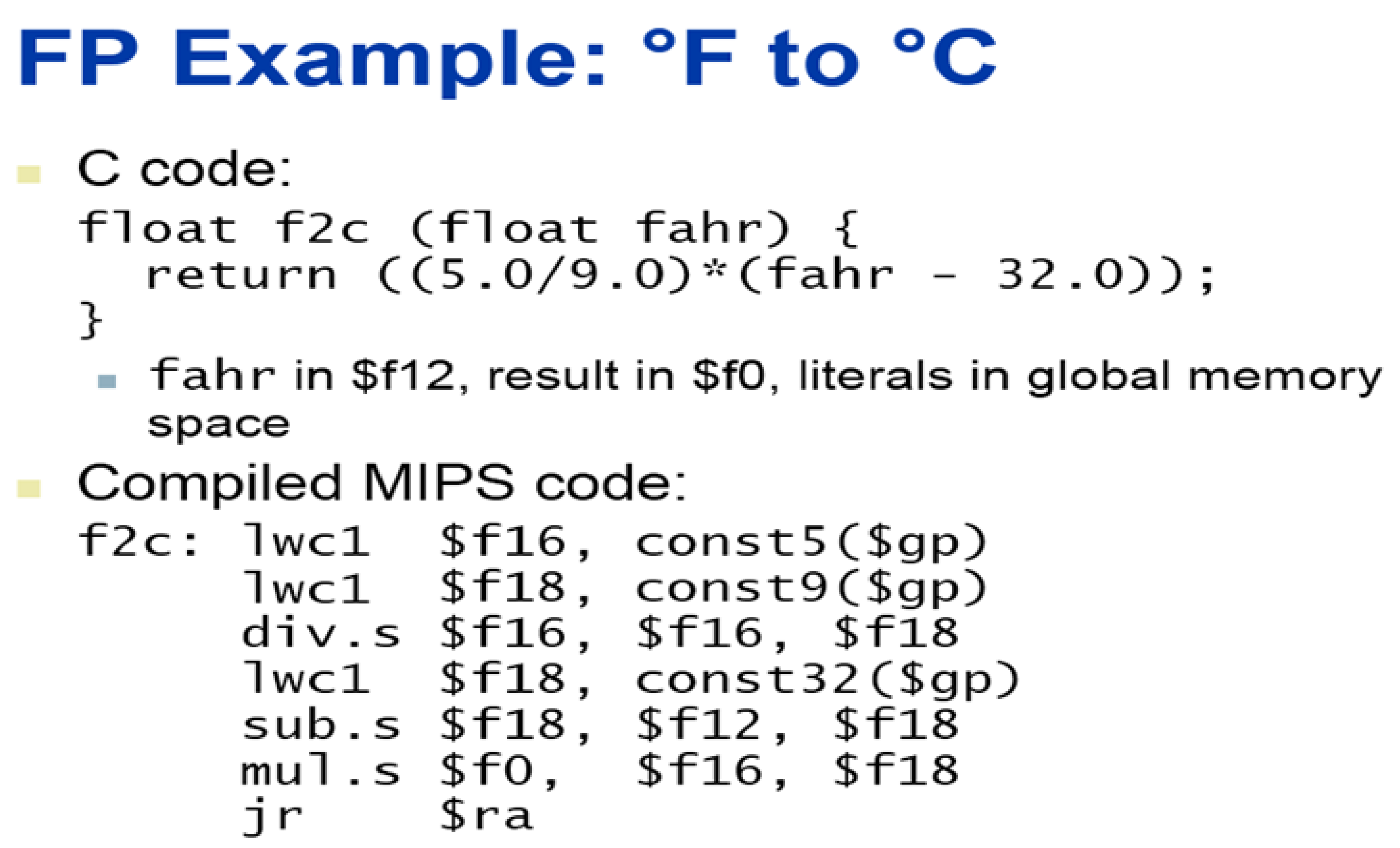
FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: \$f0, \$f1, … \$f31
  - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, …
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`



FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is `eq`, `lt`, `le`, `...`)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`



FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

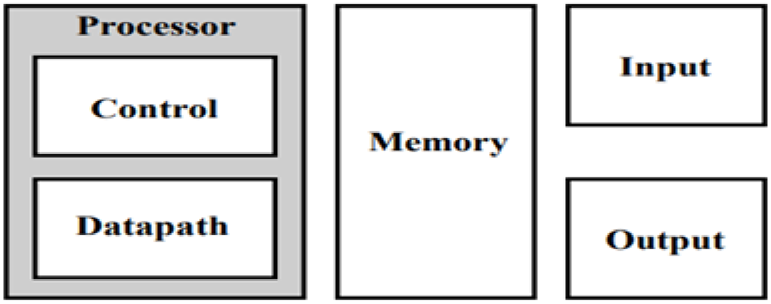
  - fahr in $f12, result in $f0, literals in global memory space
- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc1  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

Chapter Four

The Processor (Datapath and Control)

- The Five Classic Components of a Computer



➢ **Processor (CPU):** The active part of the computer that does all the work (data manipulation and decision-making)

➢ **Datapath:** Consists of the functional units of the processor. (***Show next figure***)

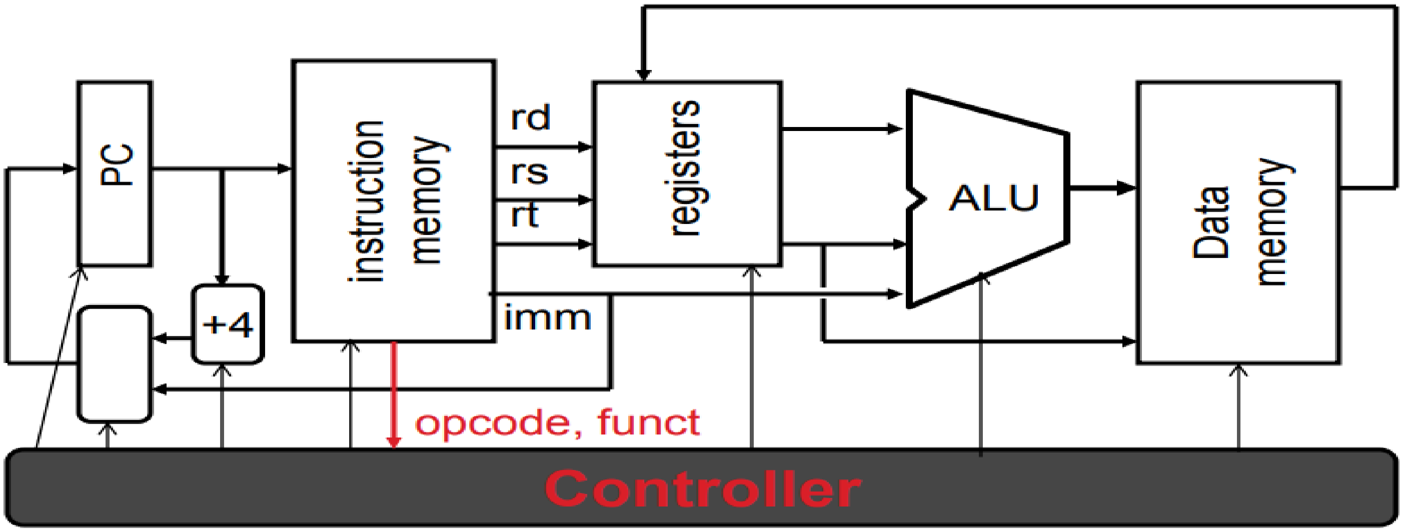
- **Elements that hold data**.
  - Program counter, register file, instruction memory, etc.
- **Elements that operate on data**.
  - ALU, adders, etc.
- **Buses for transferring data between elements**.

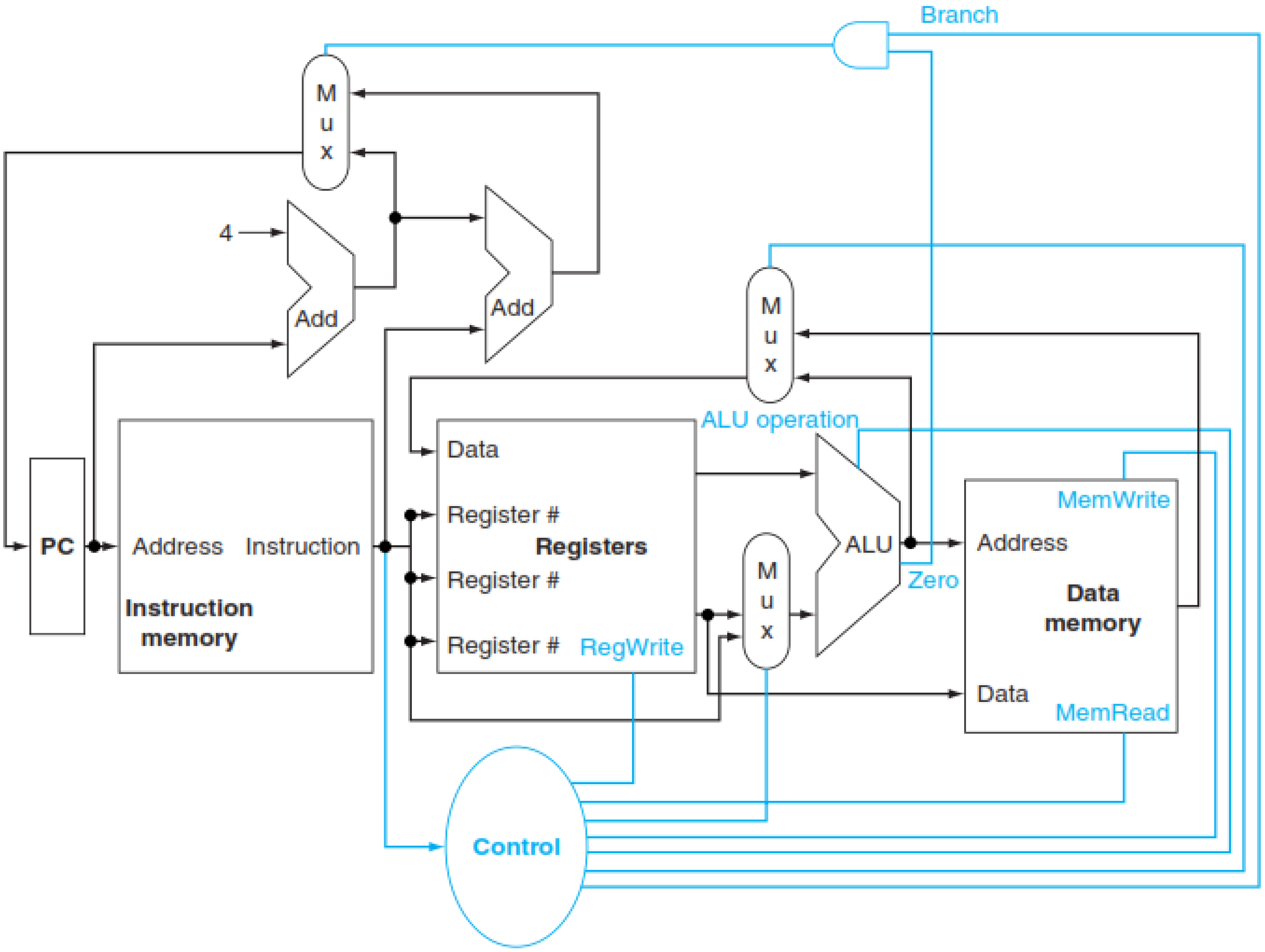
➢ **Control unit:** The **control unit** is responsible for setting all the control signals so that each instruction is executed properly.

- The control unit's input is the 32-bit instruction word.
- The outputs are values for the blue control signals in the datapath as show in fig below.
- Most of the signals can be generated from the instruction opcode alone, and not the entire 32-bit word.

Datapath and Control

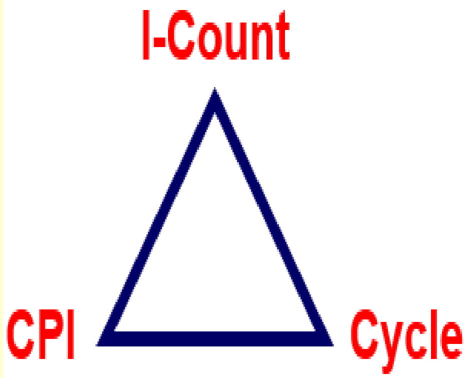
- Datapath based on data transfers required to perform instructions
- Controller causes the right transfers to happen





❑ **CPU performance factors**

- ✓ Instruction count
  - ✓ Determined by ISA and compiler
- ✓ CPI and Cycle time
  - ✓ Determined by CPU hardware



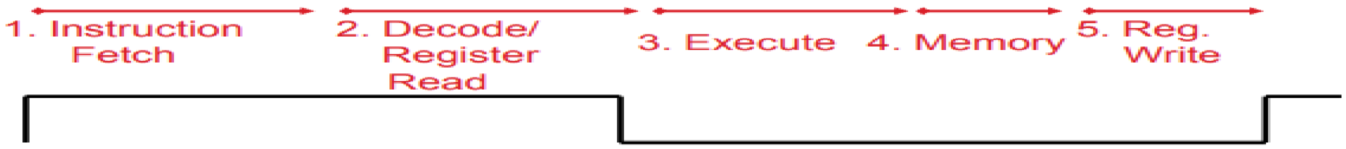
Performance = 1 / Execution time simplified to 1 / CPU execution time

CPU execution time = Instructions × CPI / (Clock rate)

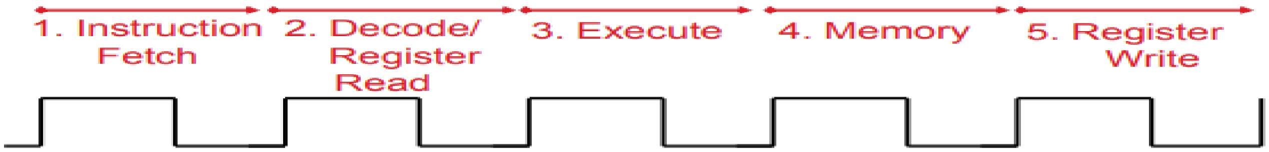
Performance = Clock rate / (Instructions × CPI)

CPU Clocking

- For each instruction, how do we control the flow of information though the datapath?
- Single Cycle CPU: All stages of an instruction completed within one long clock cycle
  - — Clock cycle sufficiently long to allow each instruction to complete all stages without interruption within one cycle



- Alternative multiple-cycle CPU: only one stage of instruction per clock cycle
  - — Clock is made as long as the slowest stage



- — Several significant advantages over single cycle execution: Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped)

Designing a Processor: Step-by-Step

- Analyze instruction set => datapath requirements
  - The meaning of each instruction is given by the register transfers
  - Datapath must include storage elements for ISA registers
  - Datapath must support each register transfer
- Select datapath components and clocking methodology
- Assemble datapath meeting the requirements
- Analyze implementation of each instruction
  - Determine the setting of control signals for register transfer
- Assemble the control logic

Review of MIPS Instruction Formats

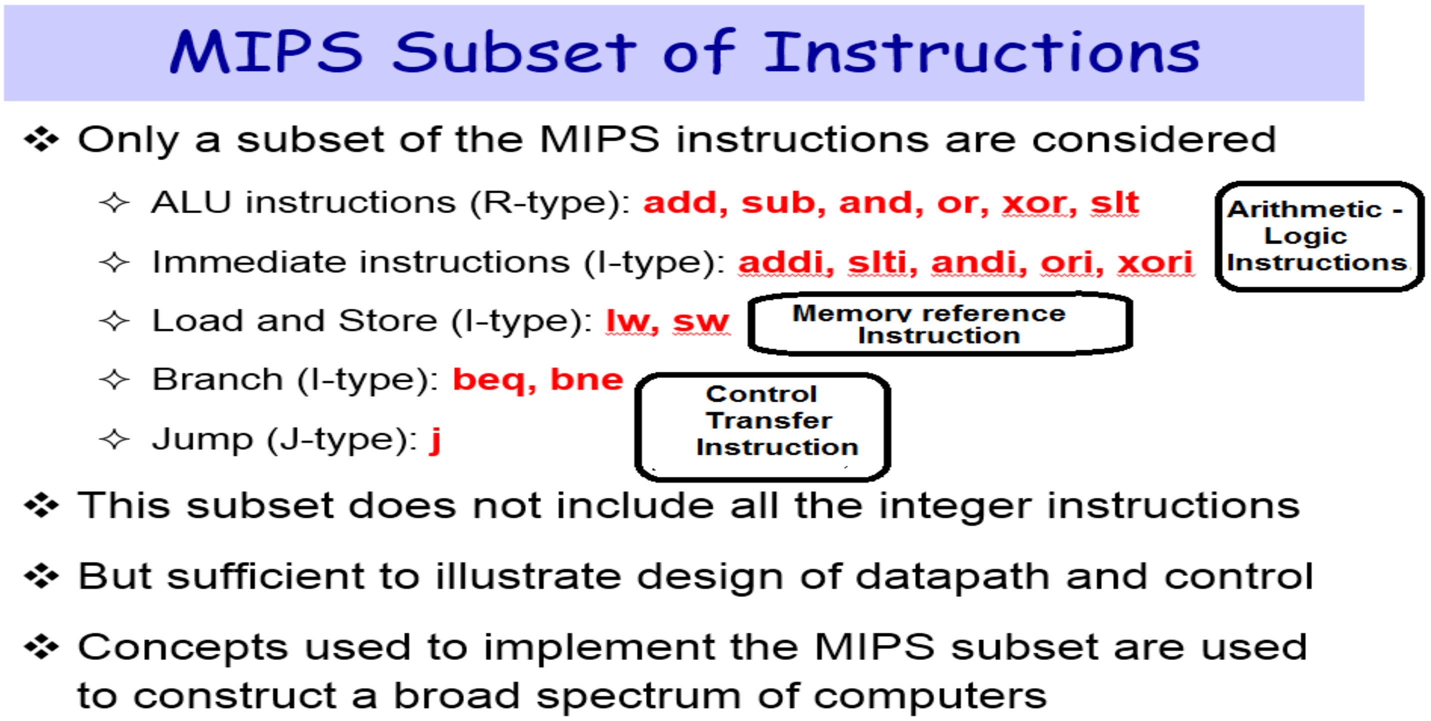
- All instructions are 32-bit wide
- Three instruction formats: R-type, I-type, and J-type

Op^6	Rs^5	Rt^5	Rd^5	sa^5	funct^6

Op^6	Rs^5	Rt^5	immediate^{16}

Op^6	immediate^{26}

  - Op^6: 6-bit opcode of the instruction
  - Rs^5, Rt^5, Rd^5: 5-bit source and destination register numbers
  - sa^5: 5-bit shift amount used by shift instructions
  - funct^6: 6-bit function field for R-type instructions
  - immediate^{16}: 16-bit immediate value or address offset
  - immediate^{26}: 26-bit target address of the jump instruction



MIPS Subset of Instructions

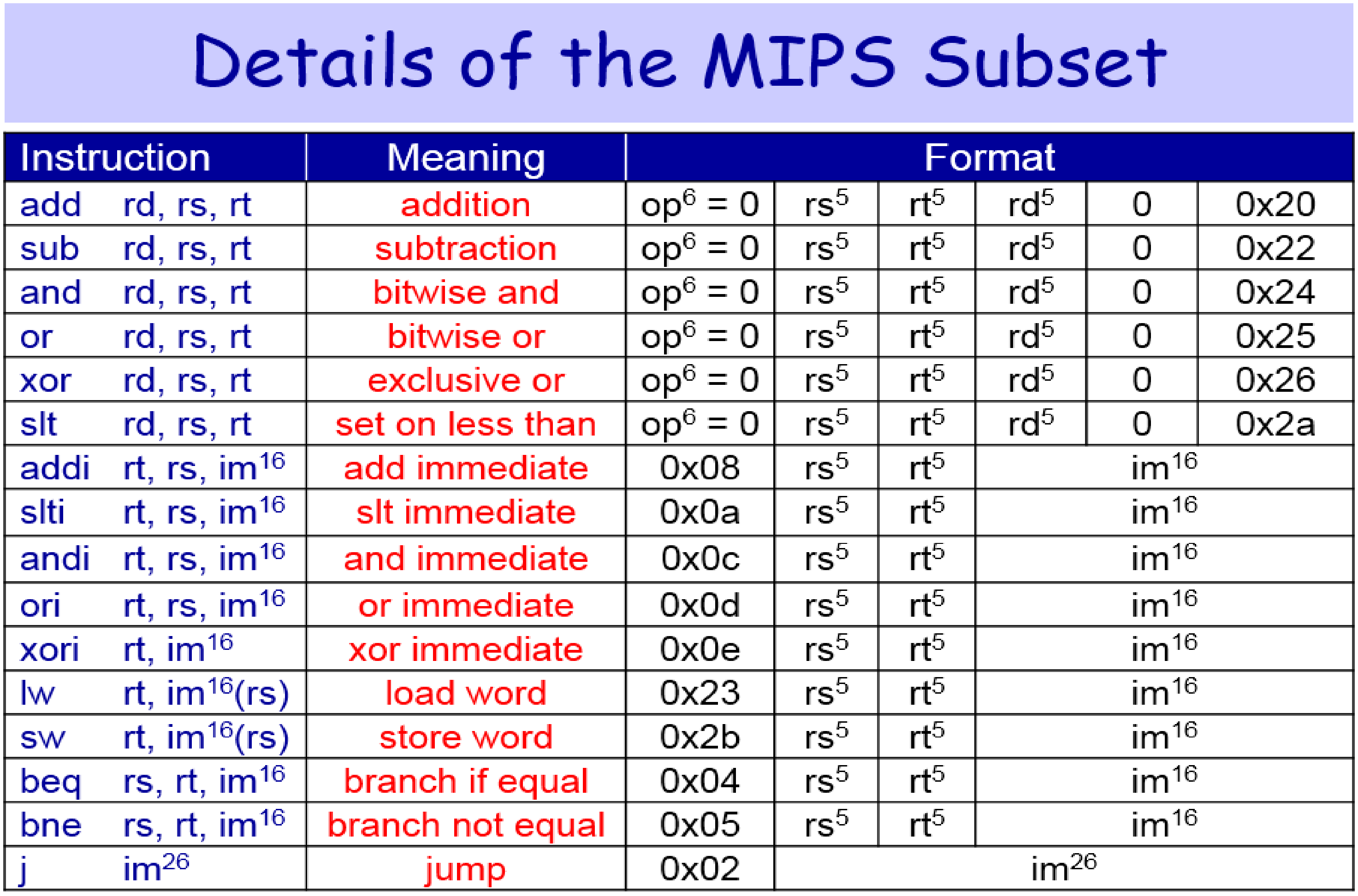
- Only a subset of the MIPS instructions are considered
  - ALU instructions (R-type): **add, sub, and, or, xor, slt**
  - Immediate instructions (I-type): **addi, slti, andi, ori, xori**
  - Load and Store (I-type): **lw, sw**
  - Branch (I-type): **beq, bne**
  - Jump (J-type): **j**

Arithmetic - Logic Instructions

Memory reference Instruction

Control Transfer Instruction

- This subset does not include all the integer instructions
- But sufficient to illustrate design of datapath and control
- Concepts used to implement the MIPS subset are used to construct a broad spectrum of computers

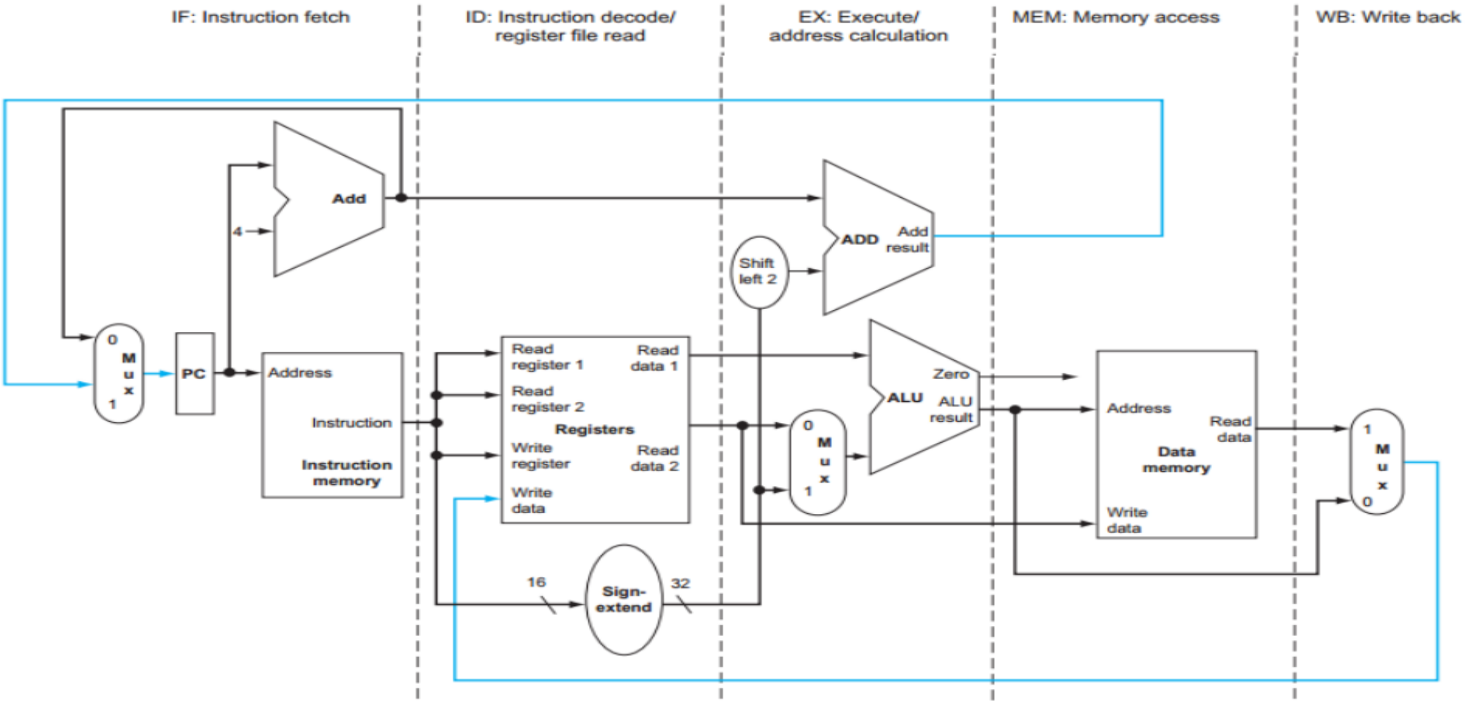


Details of the MIPS Subset

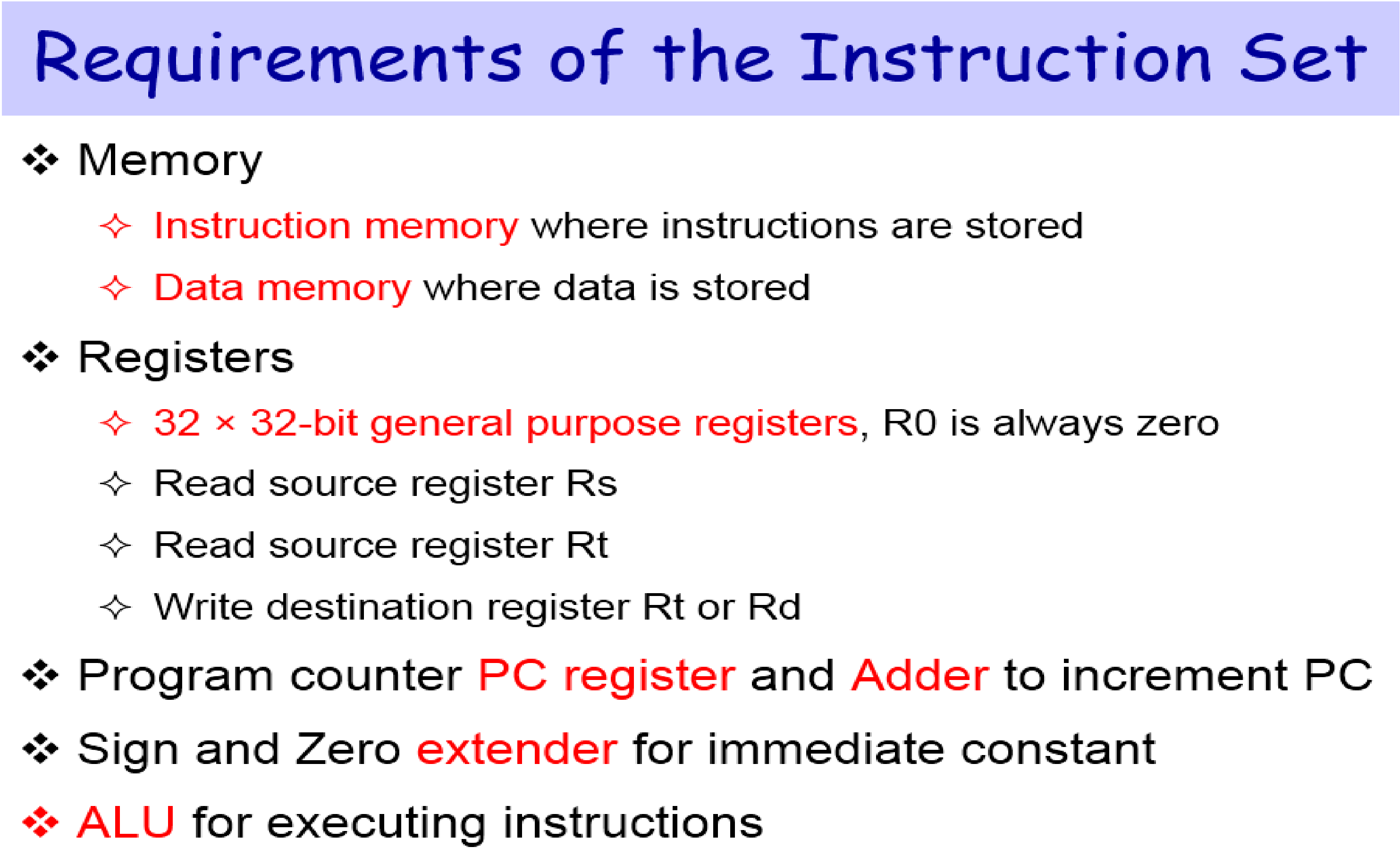
Instruction	Meaning	Format					
add rd, rs, rt	addition	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x20
sub rd, rs, rt	subtraction	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x22
and rd, rs, rt	bitwise and	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x24
or rd, rs, rt	bitwise or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x25
xor rd, rs, rt	exclusive or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x26
slt rd, rs, rt	set on less than	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2a
addi rt, rs, im^{16}	add immediate	0x08	rs^5	rt^5	im^{16}		
slti rt, rs, im^{16}	slt immediate	0x0a	rs^5	rt^5	im^{16}		
andi rt, rs, im^{16}	and immediate	0x0c	rs^5	rt^5	im^{16}		
ori rt, rs, im^{16}	or immediate	0x0d	rs^5	rt^5	im^{16}		
xori rt, im^{16}	xor immediate	0x0e	rs^5	rt^5	im^{16}		
lw rt, im^{16}(rs)	load word	0x23	rs^5	rt^5	im^{16}		
sw rt, im^{16}(rs)	store word	0x2b	rs^5	rt^5	im^{16}		
beq rs, rt, im^{16}	branch if equal	0x04	rs^5	rt^5	im^{16}		
bne rs, rt, im^{16}	branch not equal	0x05	rs^5	rt^5	im^{16}		
j im^{26}	jump	0x02	im^{26}				

Fetch/Execute Cycle

- Use the program counter (PC) to read instruction address
- Fetch the instruction from memory and increment PC
- Use fields of the instruction to select registers to read
- Execute depending on instruction class
    - Use ALU to calculate
        - ✓ Arithmetic result
        - ✓ Memory address for load/store
        - ✓ Branch target address
    - Access data memory for load/store
    - PC ← target address or PC + 4

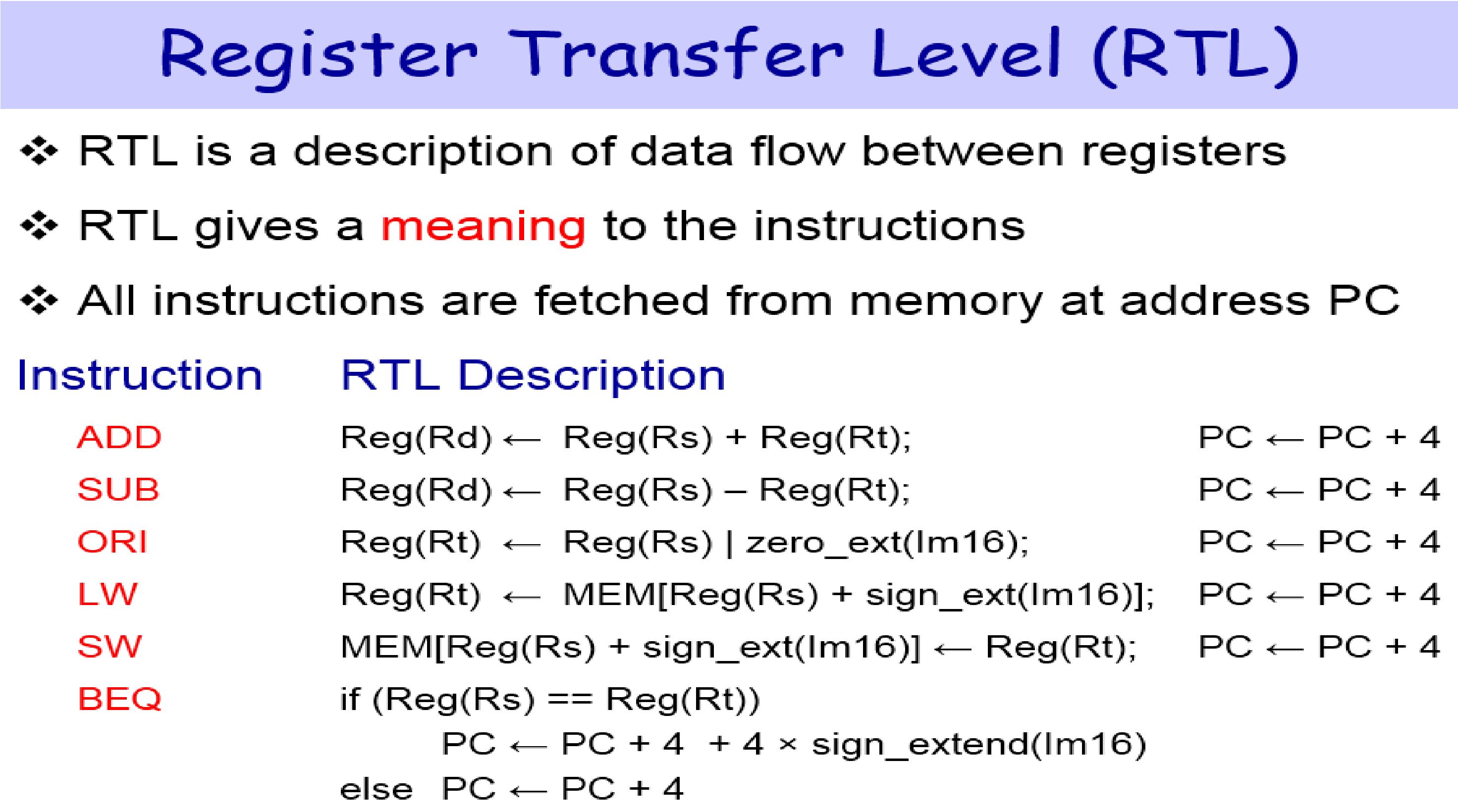


Stage	Functionality
Instruction Fetch	Send an address to the instruction memory Read the instruction (IMEM[PC])
Decode / Register Read	Generate the control signal values using the opcode & funct fields Read the register values with the relevant fields and generate the immediate
Execute	Perform arithmetic / logical operations and branch comparison
Memory	Read from / write to the data memory (DMEM)
Register Write	Write back the ALU result / the memory load / PC + 4 to the register file



Requirements of the Instruction Set

- Memory
  - Instruction memory where instructions are stored
  - Data memory where data is stored
- Registers
  - 32 × 32-bit general purpose registers, R0 is always zero
  - Read source register Rs
  - Read source register Rt
  - Write destination register Rt or Rd
- Program counter PC register and Adder to increment PC
- Sign and Zero extender for immediate constant
- ALU for executing instructions

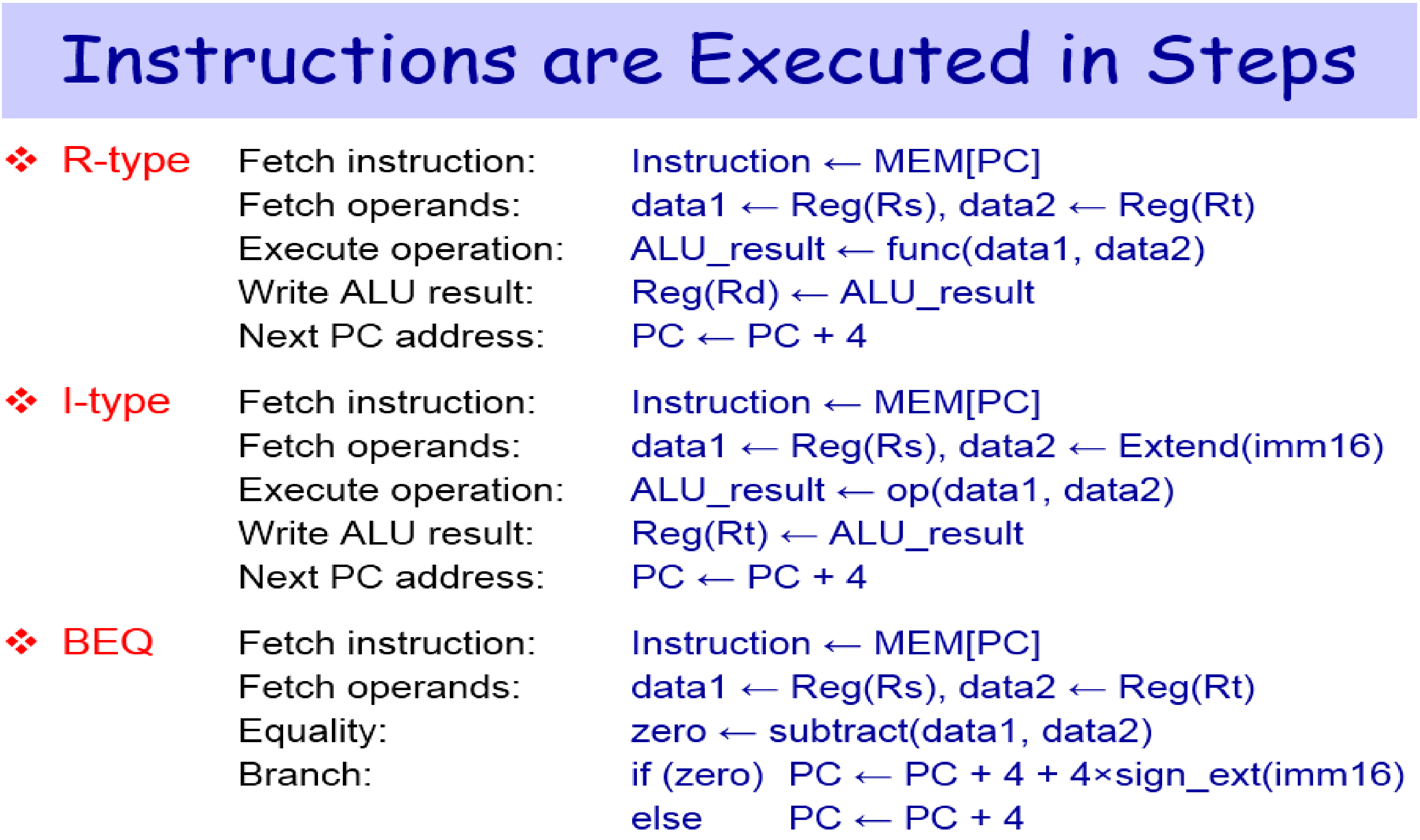


Register Transfer Level (RTL)

- RTL is a description of data flow between registers
- RTL gives a meaning to the instructions
- All instructions are fetched from memory at address PC

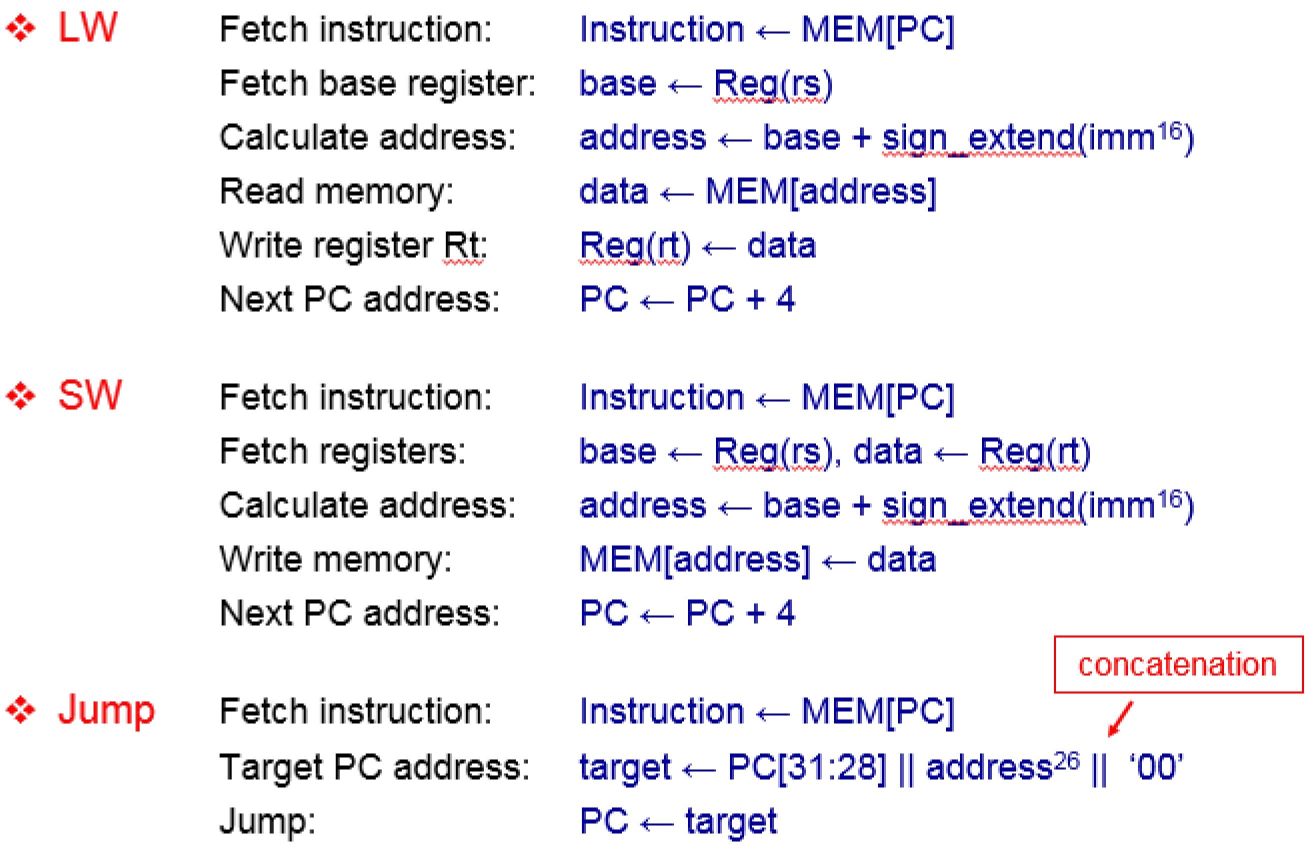
Instruction	RTL Description	
ADD	Reg(Rd) ← Reg(Rs) + Reg(Rt);	PC ← PC + 4
SUB	Reg(Rd) ← Reg(Rs) – Reg(Rt);	PC ← PC + 4
ORI	Reg(Rt) ← Reg(Rs) | zero_ext(Im16);	PC ← PC + 4
LW	Reg(Rt) ← MEM[Reg(Rs) + sign_ext(Im16)];	PC ← PC + 4
SW	MEM[Reg(Rs) + sign_ext(Im16)] ← Reg(Rt);	PC ← PC + 4

BEQ	if (Reg(Rs) == Reg(Rt))
	PC ← PC + 4 + 4 × sign_extend(Im16)
	else PC ← PC + 4	



Instructions are Executed in Steps

❖ R-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
	Execute operation:	ALU_result ← func(data1, data2)
	Write ALU result:	Reg(Rd) ← ALU_result
	Next PC address:	PC ← PC + 4
❖ I-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(Rs), data2 ← Extend(imm16)
	Execute operation:	ALU_result ← op(data1, data2)
	Write ALU result:	Reg(Rt) ← ALU_result
	Next PC address:	PC ← PC + 4
❖ BEQ	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
	Equality:	zero ← subtract(data1, data2)
	Branch:	if (zero) PC ← PC + 4 + 4×sign_ext(imm16)
		else PC ← PC + 4



❖ LW	Fetch instruction:	Instruction ← MEM[PC]
	Fetch base register:	base ← Reg(rs)
	Calculate address:	address ← base + sign_extend(imm^{16})
	Read memory:	data ← MEM[address]
	Write register Rt:	Reg(rt) ← data
	Next PC address:	PC ← PC + 4
❖ SW	Fetch instruction:	Instruction ← MEM[PC]
	Fetch registers:	base ← Reg(rs), data ← Reg(rt)
	Calculate address:	address ← base + sign_extend(imm^{16})
	Write memory:	MEM[address] ← data
	Next PC address:	PC ← PC + 4
❖ Jump	Fetch instruction:	Instruction ← MEM[PC]
	Target PC address:	target ← PC[31:28] || $address^{26}$ || ‘00’
	Jump:	PC ← target

concatenation

MIPS Implementations

Two MIPS implementations will be studied

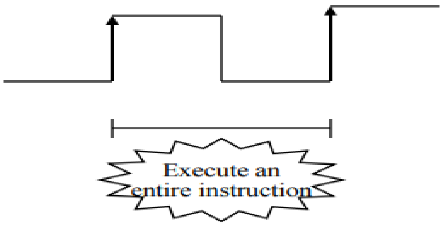
- ■ ***A simplified version***
- ■ ***A more realistic pipelined version***

Any instruction set can be implemented in many different ways:-

✓ **Single cycle**: All "steps" of executing an instruction are done in one clock cycle. The cycle is long to accommodate longest path.

✧ ***Advantage***: One clock cycle per instruction

✧ ***Disadvantage***: long cycle time



✓ **Multi cycle**: steps (cycles) to execute instruction.

✧ break fetch/execute cycle into multiple steps

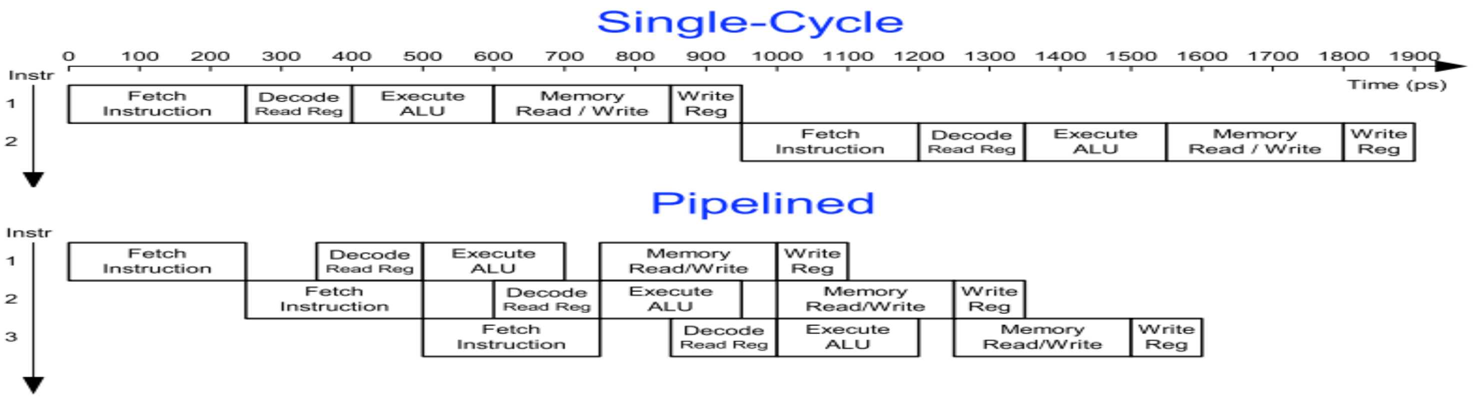
✧ perform 1 step in each clock cycle

✓ **Pipelining** lets a processor overlap the execution of several instructions, potentially leading to big performance gains.

✧ execute each instruction in multiple steps

✧ perform 1 step / instruction in each clock cycle

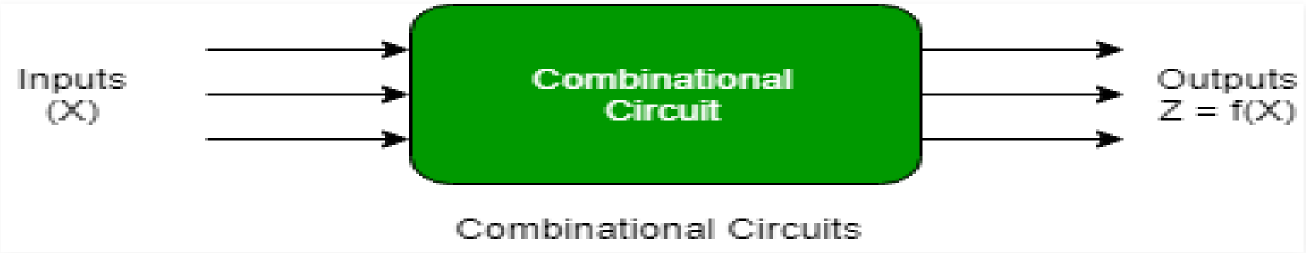
✧ process multiple instructions in parallel



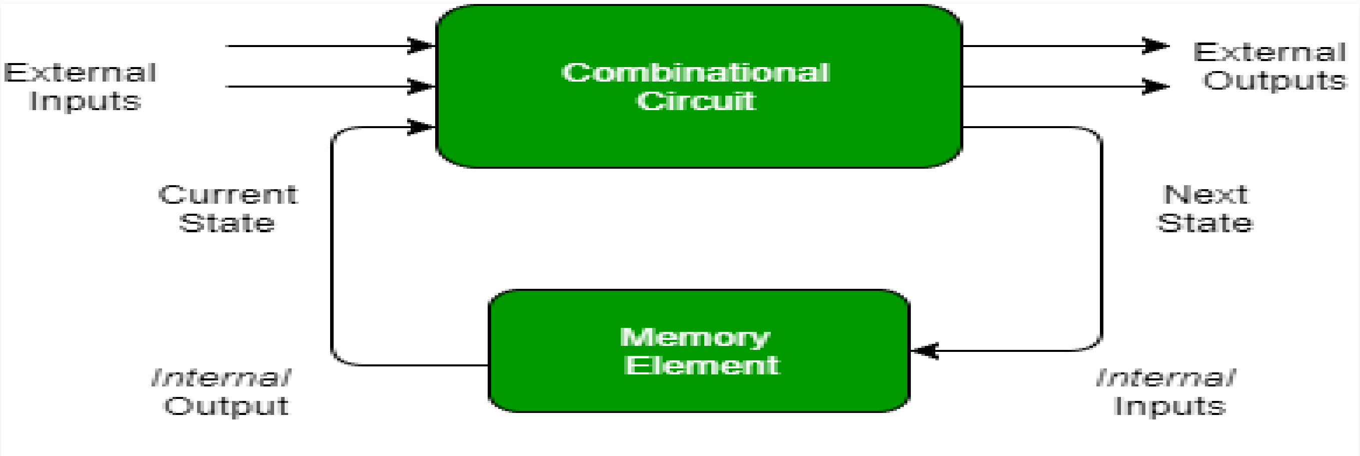
Logic Design Basics

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Review: Two Types of Logic Components



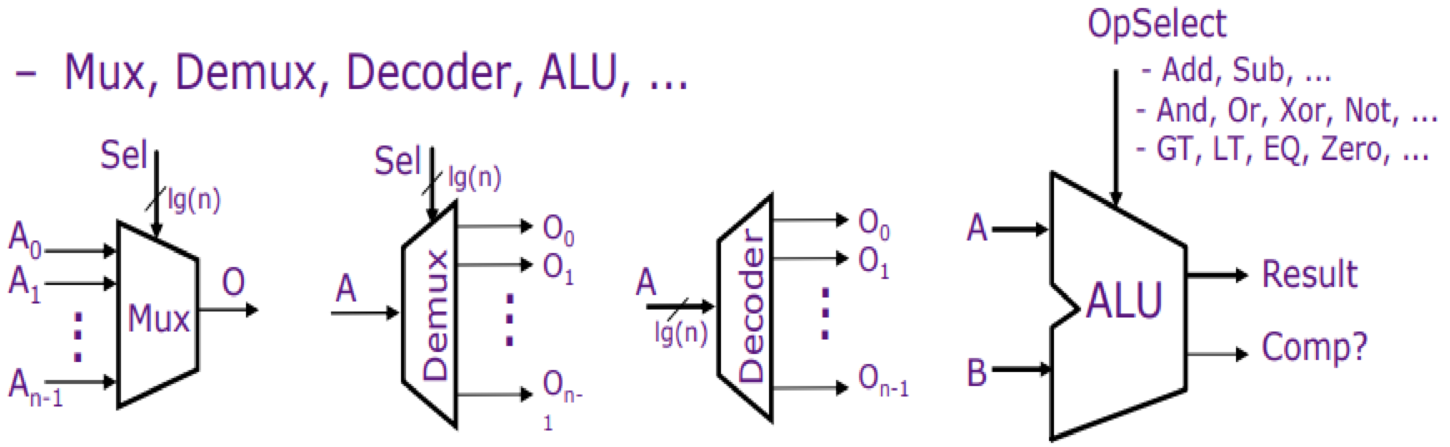
Combinational Circuits



Sequential Circuit

Combinational circuits

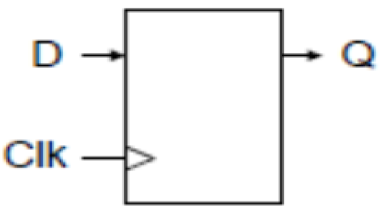
- Mux, Demux, Decoder, ALU, ...

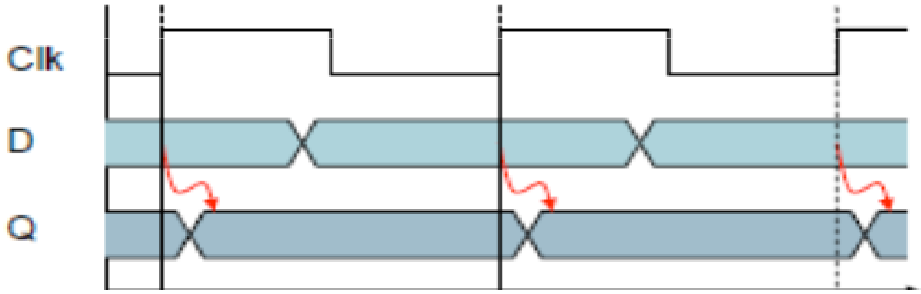


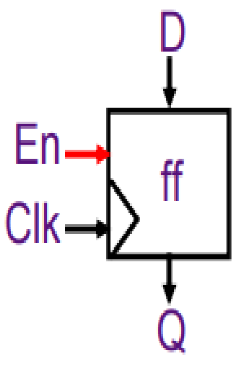
Sequential Elements

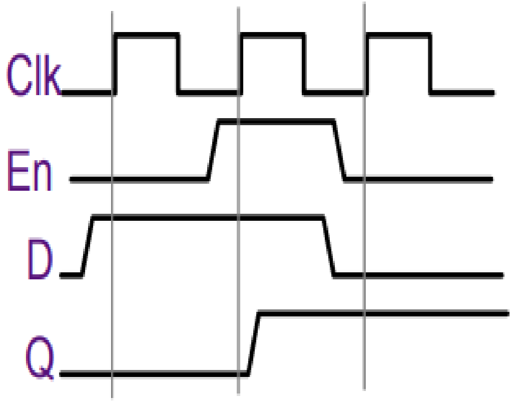
(Flipflop, Register, Register file, SRAM, DRAM)

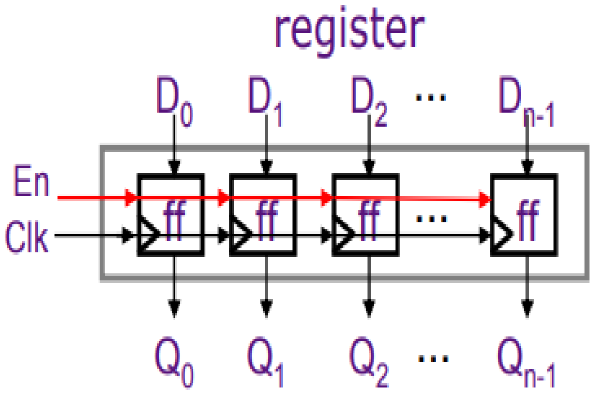
- **<u>Registers</u>** are implemented with arrays of D-flipflops
- Registers contain (store) data
- Uses a clock signal to determine when to update the stored value
- Edge-triggered: update when CLK changes from 0 to 1
- All state elements together define the state of the machine



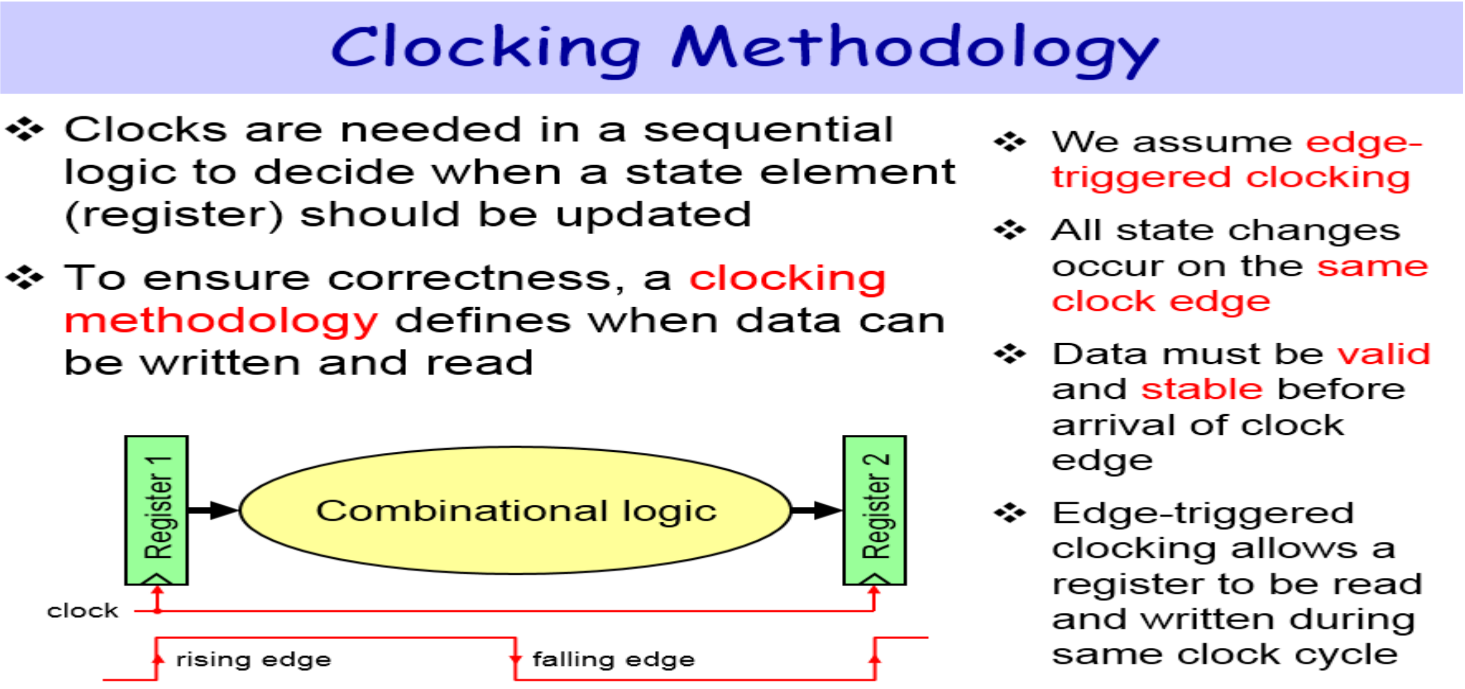








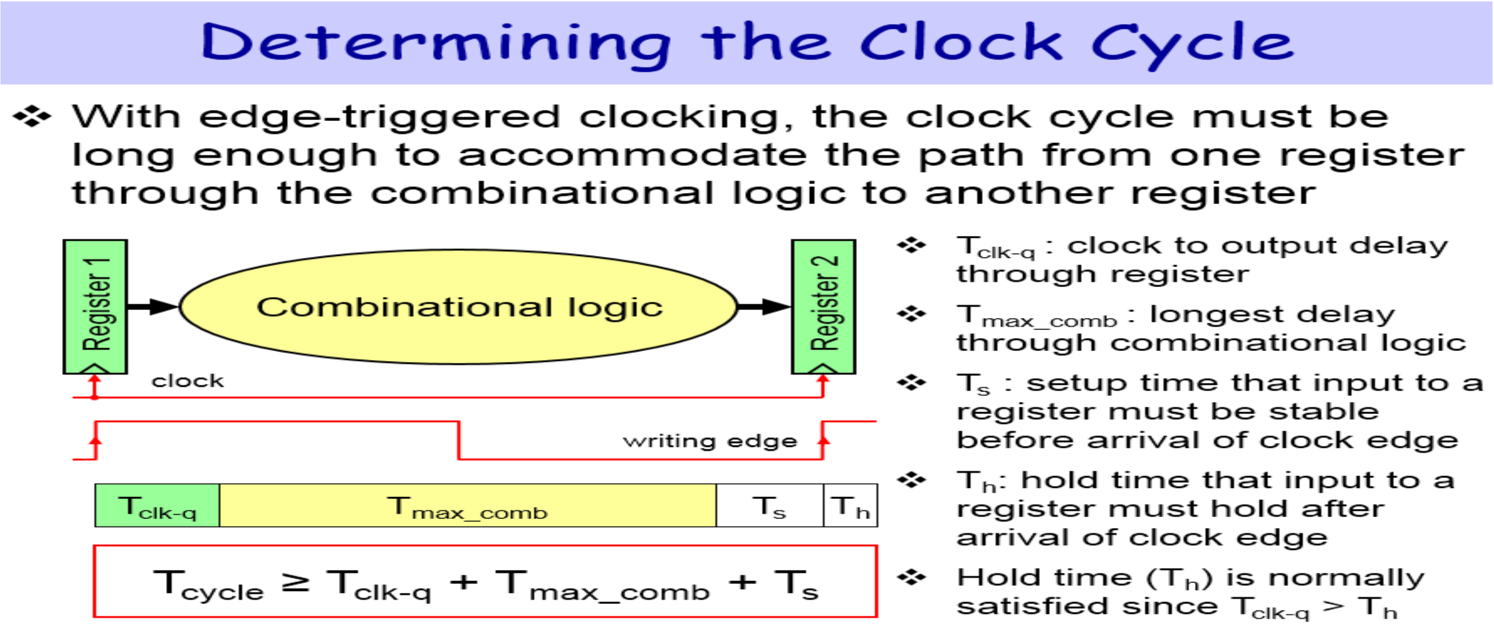
Clocking methodology :
The approach used to determine when data is valid and stable relative to the clock.



Clocking Methodology

- Clocks are needed in a sequential logic to decide when a state element (register) should be updated
- To ensure correctness, a clocking methodology defines when data can be written and read

- We assume edge-triggered clocking
- All state changes occur on the same clock edge
- Data must be valid and stable before arrival of clock edge
- Edge-triggered clocking allows a register to be read and written during same clock cycle



Determining the Clock Cycle

- With edge-triggered clocking, the clock cycle must be long enough to accommodate the path from one register through the combinational logic to another register

$$T_{cycle} \geq T_{clk-q} + T_{max\_comb} + T_s$$

- T_{clk-q} : clock to output delay through register
- T_{max_comb} : longest delay through combinational logic
- T_s : setup time that input to a register must be stable before arrival of clock edge
- T_h: hold time that input to a register must hold after arrival of clock edge
- Hold time (T_h) is normally satisfied since $T_{clk-q} > T_h$

Single-cycle Implementation

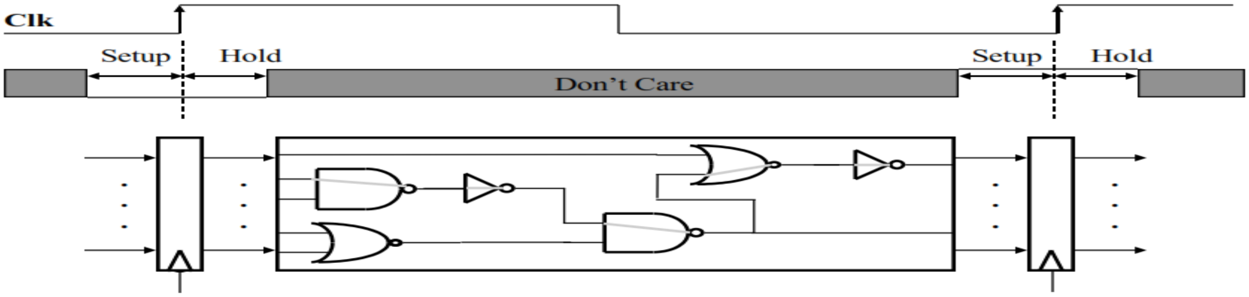
First Step: Building a Datapath

Next Step: Implementing Control

- Use a single long clock cycle for every instruction.
- This approach is much slower than a *multi-cycle* implementation where different instruction classes can take different numbers of cycles.
    - In a single-cycle implementation, every instruction must take the same amount of time as the slowest instruction take the same amount of time as the slowest instruction.
    - In a multi-cycle implementation this problem is avoided by allowing quicker instructions to use fewer cycles.

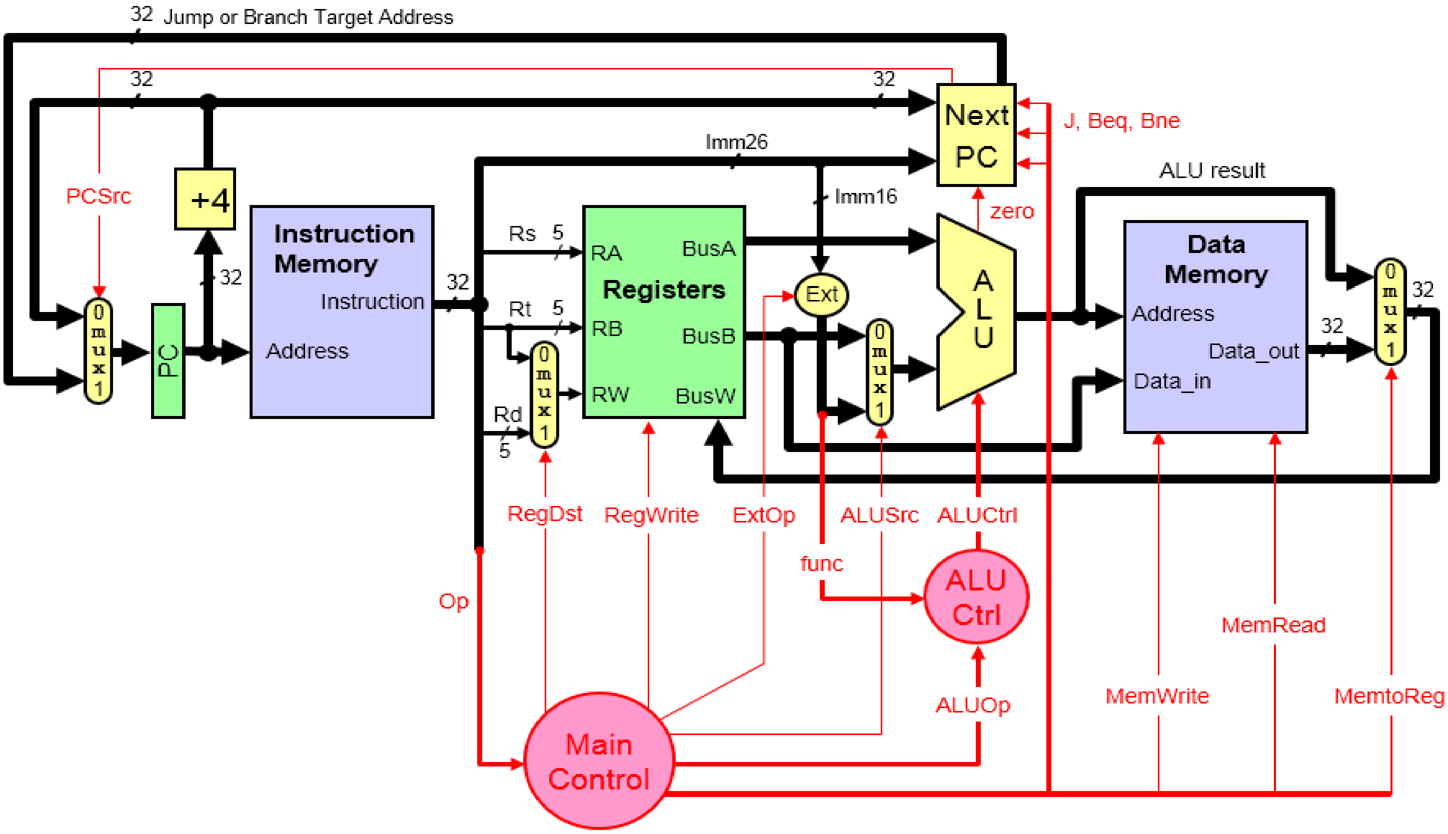
MIPS makes it easier

- Instructions same size
- Source registers always in same place
- Immediates same size, location
- Operations always on registers/immediates



Single cycle datapath => CPI=1, CCT => long

Single Cycle Datapath and Control



Note

The single-cycle datapath conceptually described in this section *must* have separate instruction and data memories because

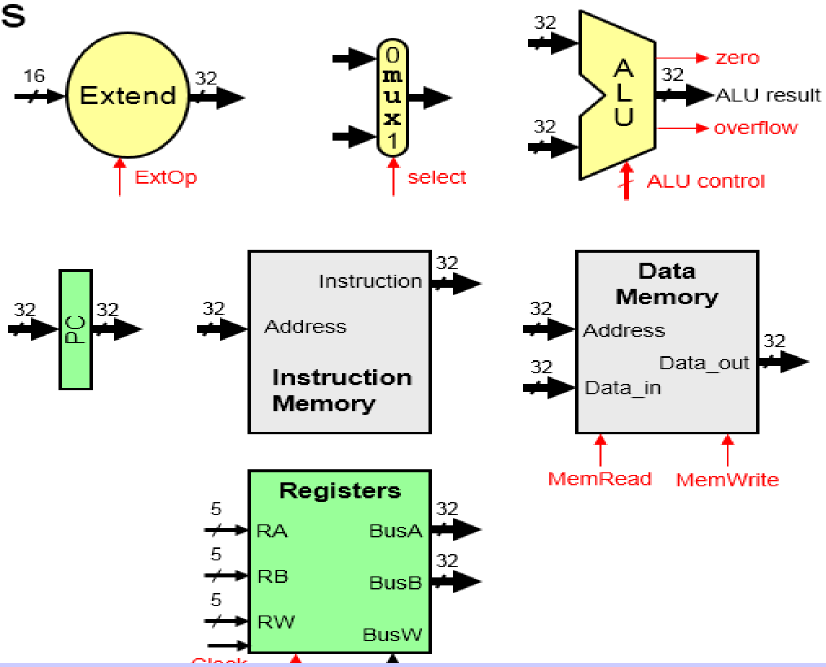
1. The format of data and instructions is different in MIPS and hence different memories are needed.
2. Having separate memories is less expensive.
3. The processor operates in one cycle and cannot use a single-ported memory for two different accesses within that cycle.

Datapath consists of the functional units of the processor.

- Elements that hold data**:** Program counter, register file, instruction memory, etc.
- Elements that operate on data**:** ALU, adders, etc.
- Buses for transferring data between elements.

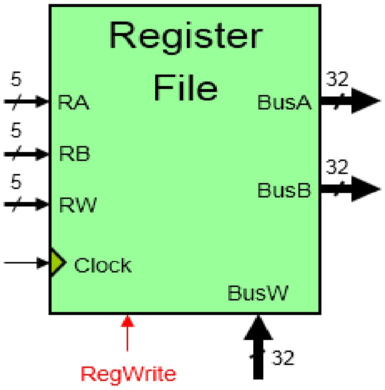
Components of the Datapath

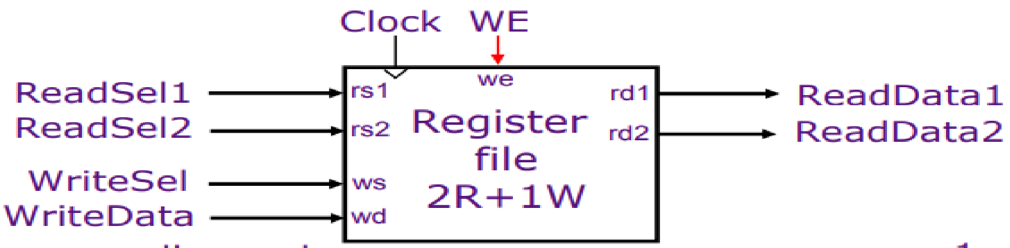
- ❖ Combinational Elements
  - ✧ ALU, Adder
  - ✧ Immediate extender
  - ✧ Multiplexers
- ❖ Storage Elements
  - ✧ Instruction memory
  - ✧ Data memory
  - ✧ PC register
  - ✧ Register file
- ❖ Clocking methodology

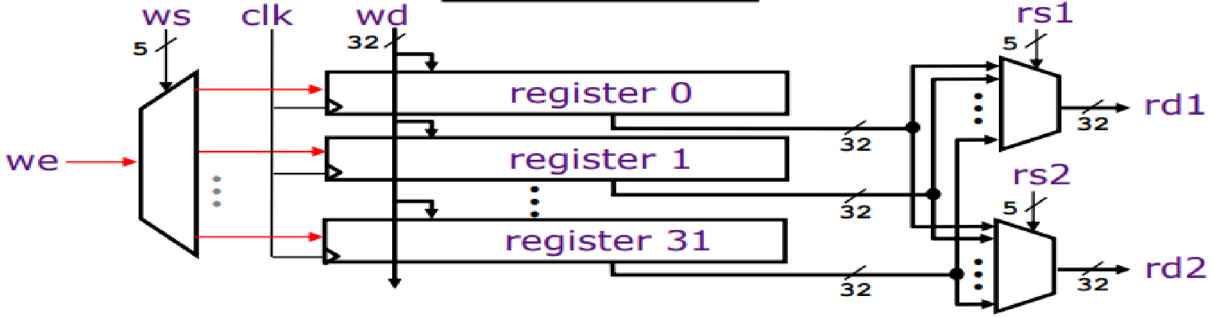


MIPS Register File

- ❖ Register File consists of 32 × 32-bit registers
  - ✧ BusA and BusB: 32-bit output busses for reading 2 registers
  - ✧ BusW: 32-bit input bus for writing a register when RegWrite is 1
  - ✧ Two registers read and one written in a cycle
- ❖ Registers are selected by:
  - ✧ RA selects register to be read on BusA
  - ✧ RB selects register to be read on BusB
  - ✧ RW selects the register to be written



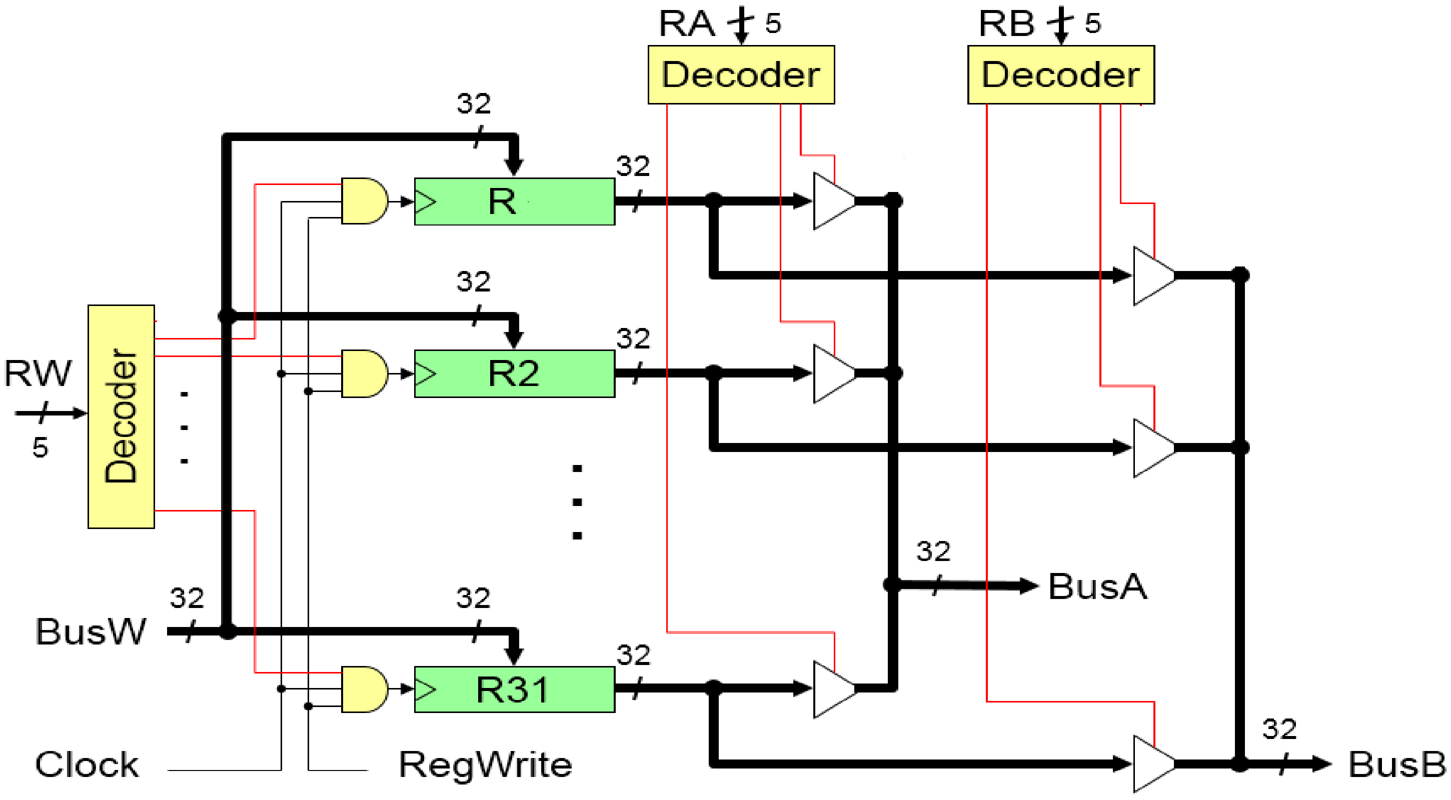


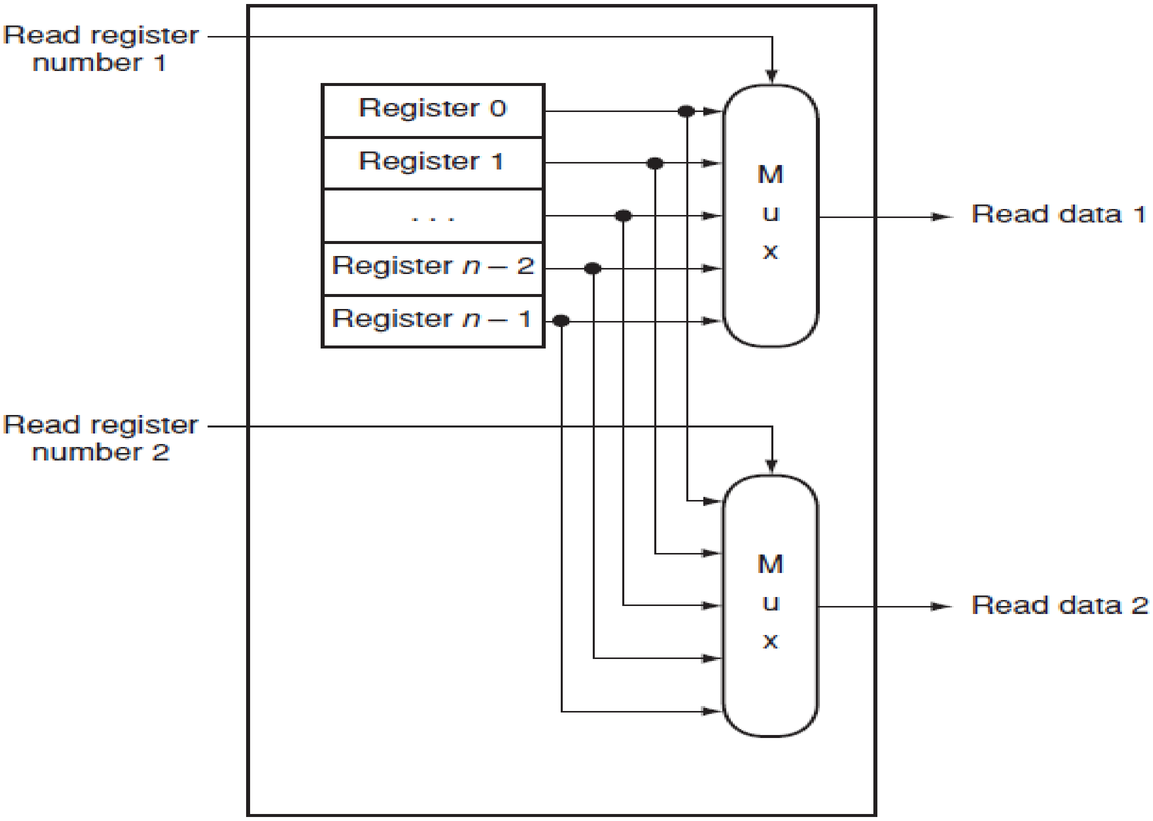


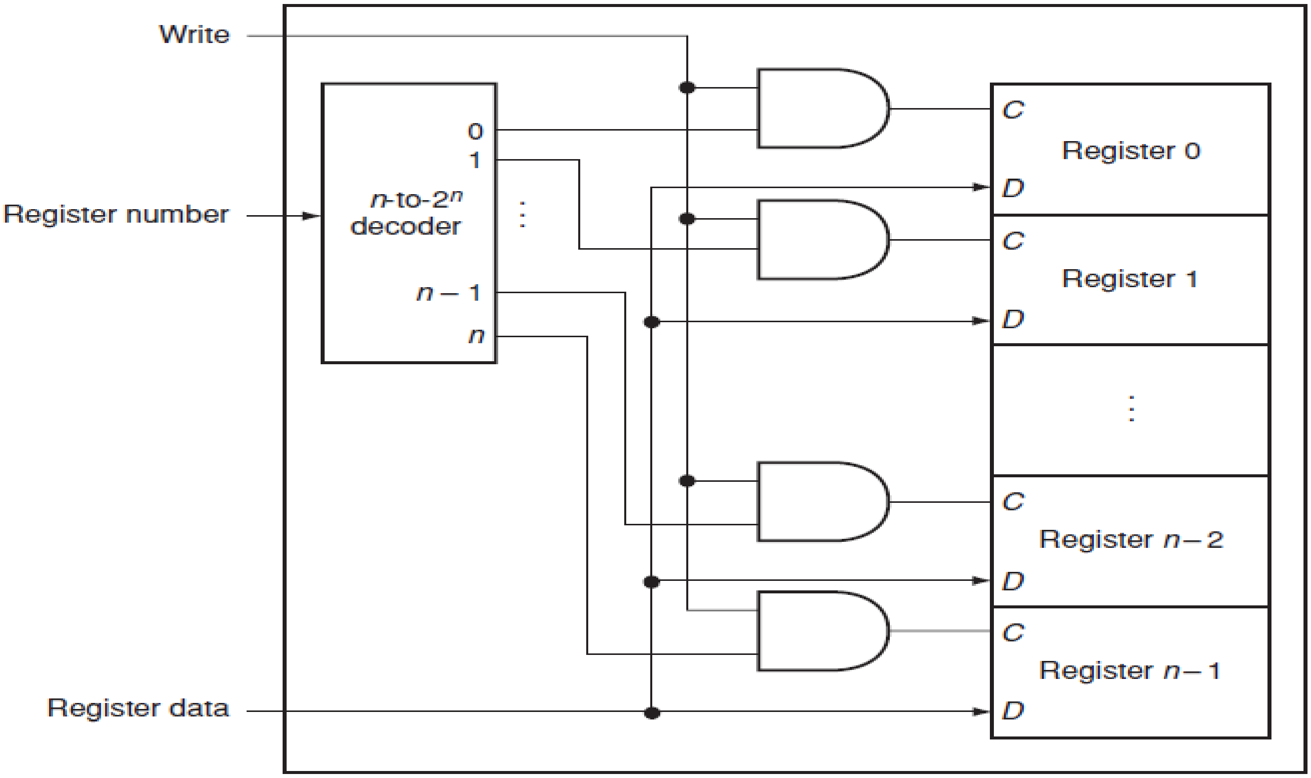
- No timing issues in reading a selected register
- Register files with a large number of ports are difficult to design

- a Read can be done any time (i.e. combinational)
- a Write is performed at the rising clock edge if it is enabled

  ⇒ *the write address and data must be stable at the clock edge*




Read register number 1
Register 0
Register 1
. . .
Register $n-2$
Register $n-1$
M u x
Read data 1
Read register number 2
M u x
Read data 2

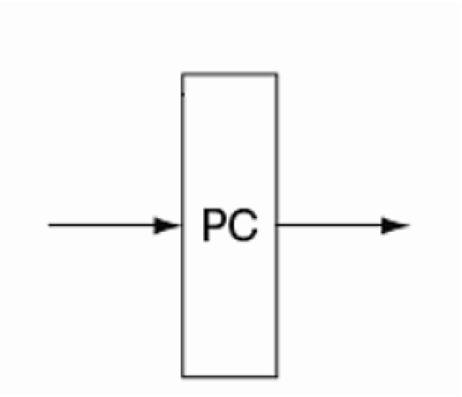

Write
Register number
0
1
n-to-2^n decoder
$n-1$
n
Register data
C
Register 0
D
C
Register 1
D
C
Register $n-2$
D
C
Register $n-1$
D

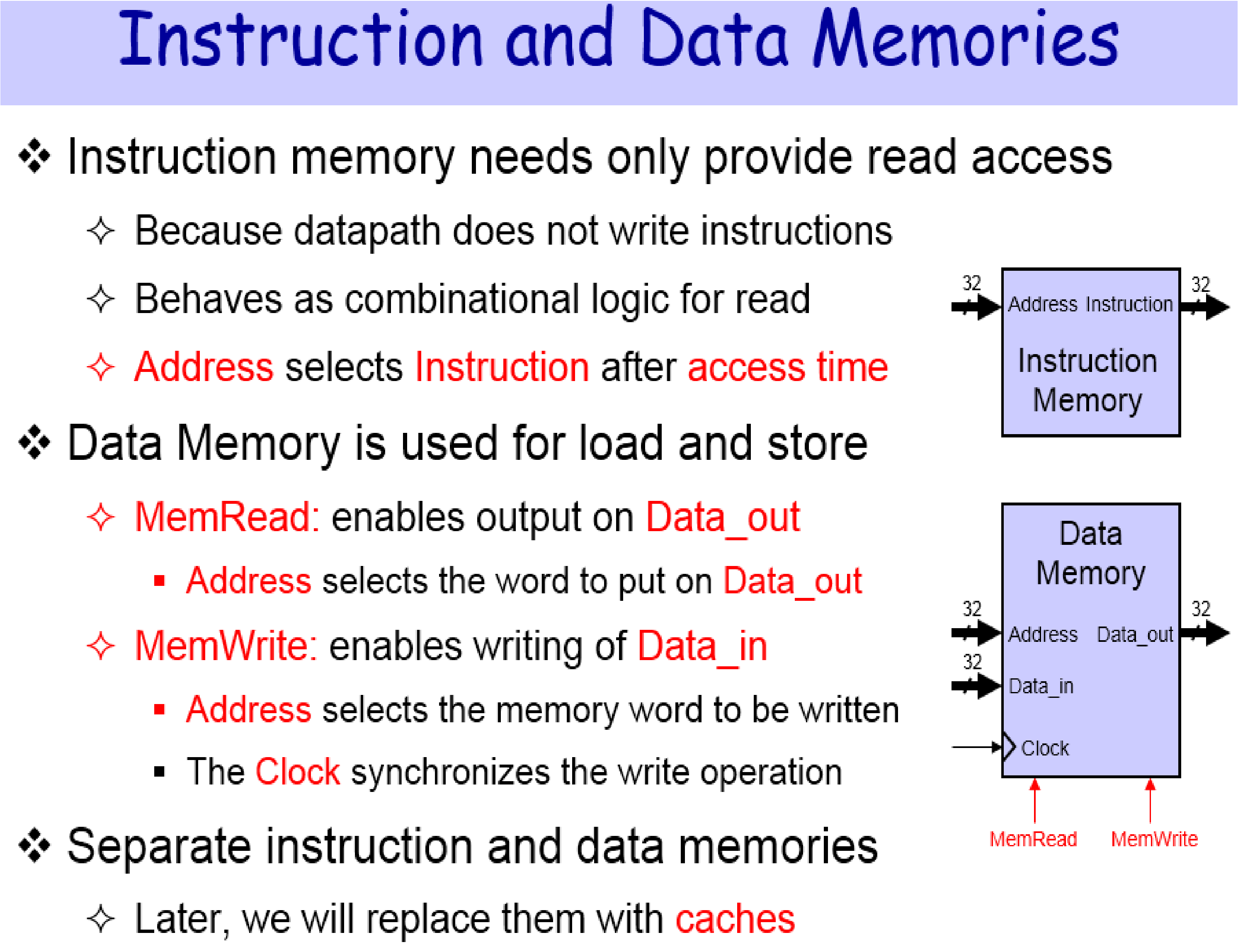
Next, we have the *program counter* or *PC*.

The PC is a state element that holds the address of the current instruction. Essentially, it is just a 32-bit register which holds the instruction address and is updated at the end of every clock cycle.

- Normally PC increments sequentially except for branch instructions

The arrows on either side indicate that the PC state element is both readable and writeable.

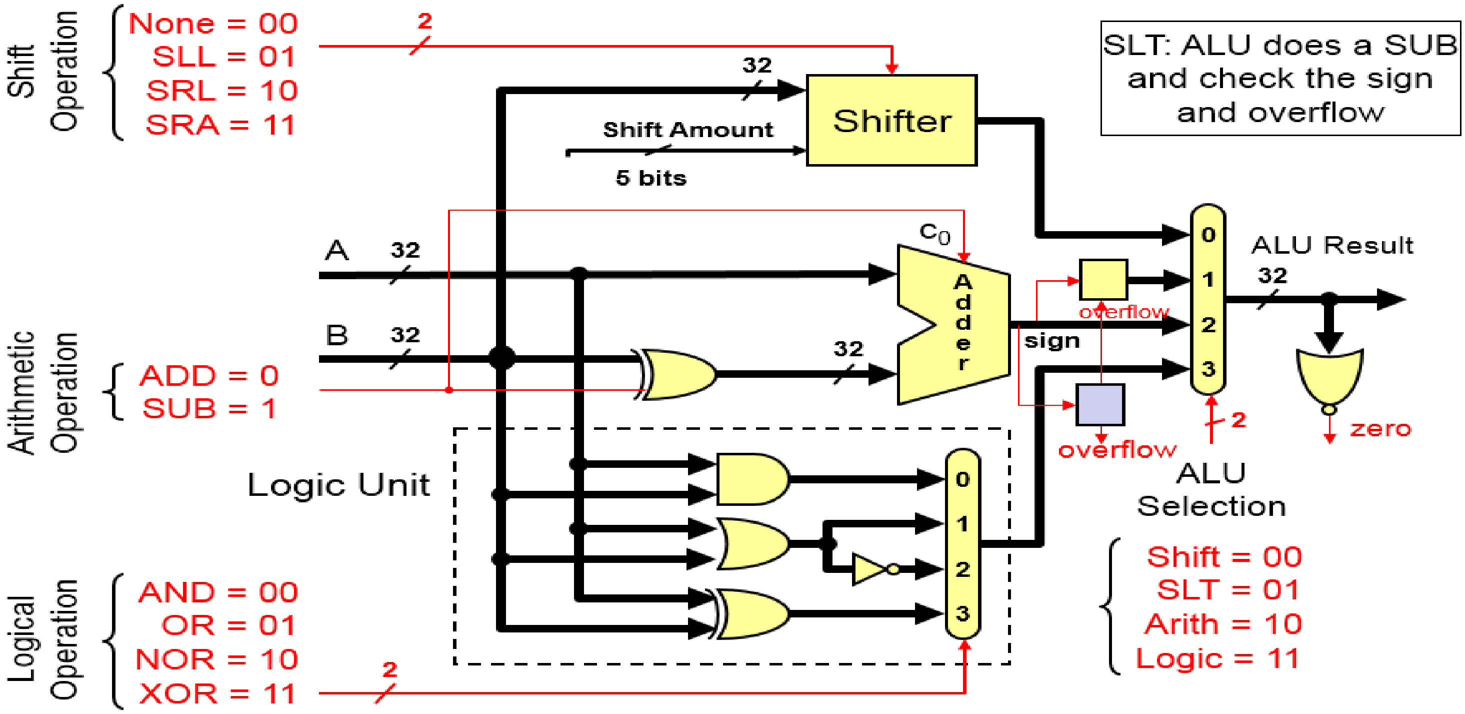




Instruction and Data Memories

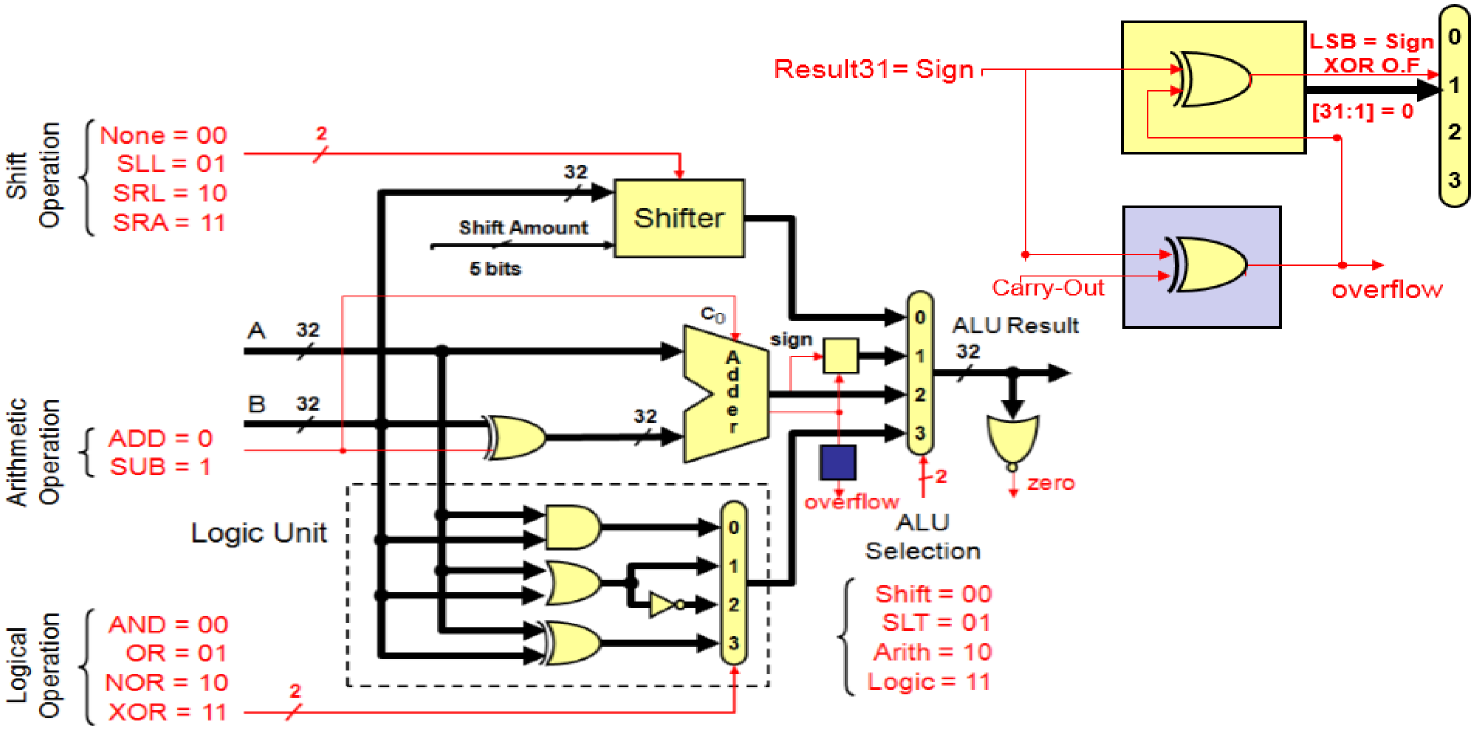
- Instruction memory needs only provide read access
  - Because datapath does not write instructions
  - Behaves as combinational logic for read
  - Address selects Instruction after access time
- Data Memory is used for load and store
  - MemRead: enables output on Data_out
    - Address selects the word to put on Data_out
  - MemWrite: enables writing of Data_in
    - Address selects the memory word to be written
    - The Clock synchronizes the write operation
- Separate instruction and data memories
  - Later, we will replace them with caches

Building a Multifunction ALU



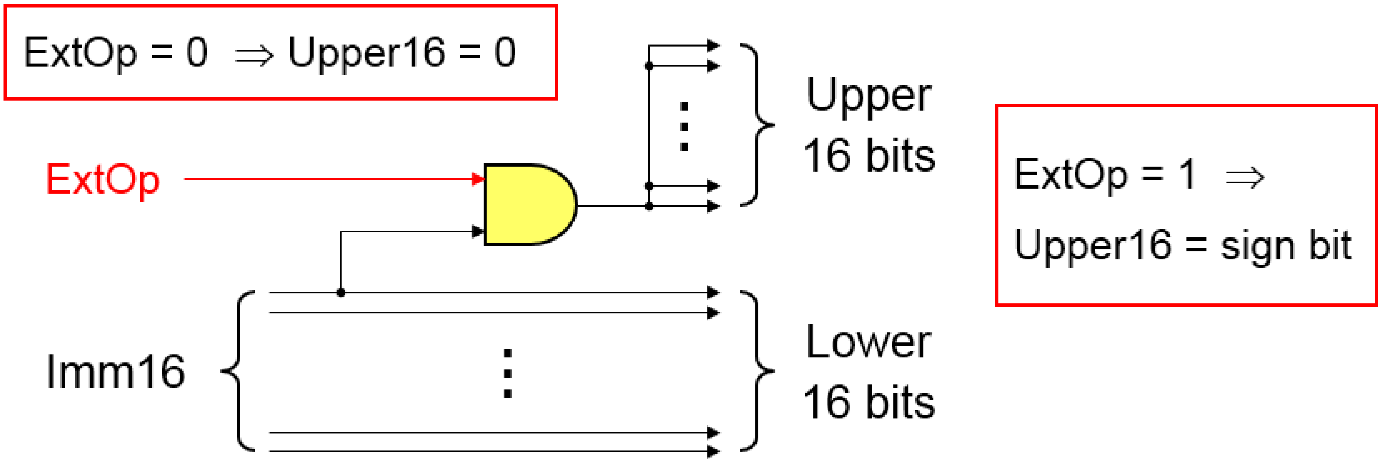
SLT: ALU does a SUB and check the sign and overflow

Overflow and SLT



Details of the Extender

- ❖ Two types of extensions
  - ✧ Zero-extension for unsigned constants
  - ✧ Sign-extension for signed constants
- ❖ Control signal ExtOp indicates type of extension
- ❖ Extender Implementation: wiring and one AND gate



ExtOp = 0 ⇒ Upper16 = 0

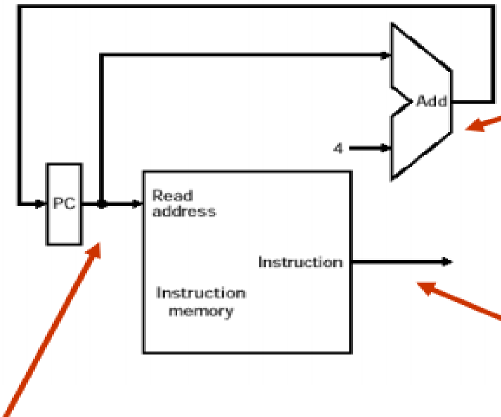
ExtOp = 1 ⇒ Upper16 = sign bit

So now we have instruction memory, PC, and adder datapath elements. Now, we can talk about the general steps taken to execute a program.

- Instruction fetching: use the address in the PC to fetch the current instruction from instruction memory.
- Instruction decoding: determine the fields within the instruction
- Instruction execution: perform the operation indicated by the instruction.
- Update the PC to hold the address of the next instruction

Instruction Fetching Datapath

- We can now assemble the datapath from its components
- For instruction fetching, we need …
  - Program Counter (PC) register
  - Instruction Memory
  - Adder for incrementing PC

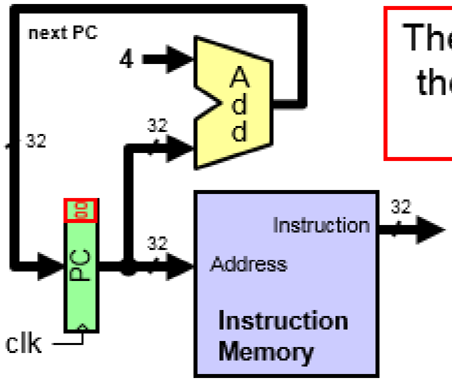


Increment PC by four after reading current instruction - by four since an instruction is 32 bits long

Instruction read from the memory - send to rest of the data path

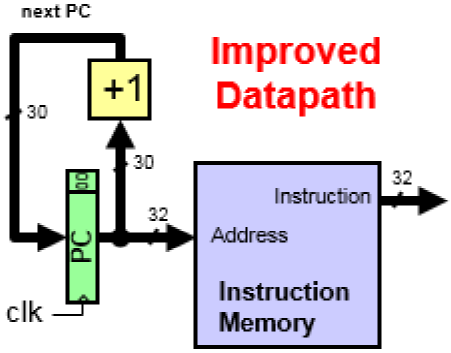
Send address from PC to the instruction memory to read the instruction at IM[PC]

Improved datapath increments upper 30 bits of PC by 1



The least significant 2 bits of the PC are '00' since PC is a multiple of 4

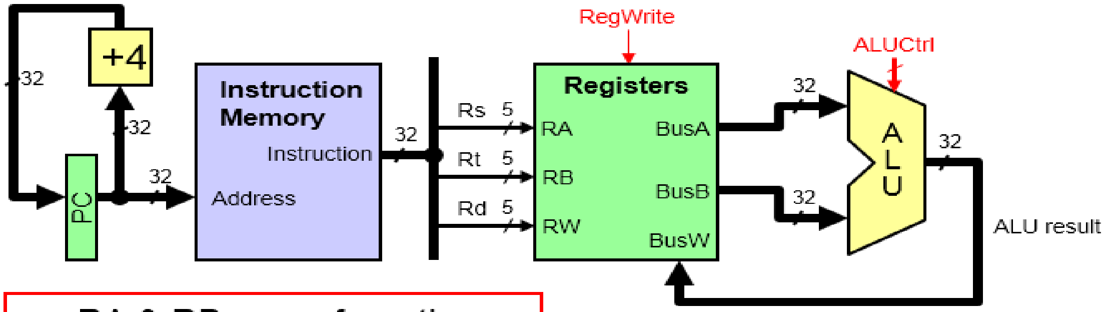
Datapath does not handle branch or jump instructions



Improved Datapath

Datapath for R-type Instructions

Op^6	Rs^5	Rt^5	Rd^5	sa^5	funct^6



RA & RB come from the instruction's Rs & Rt fields

ALU inputs come from BusA & BusB

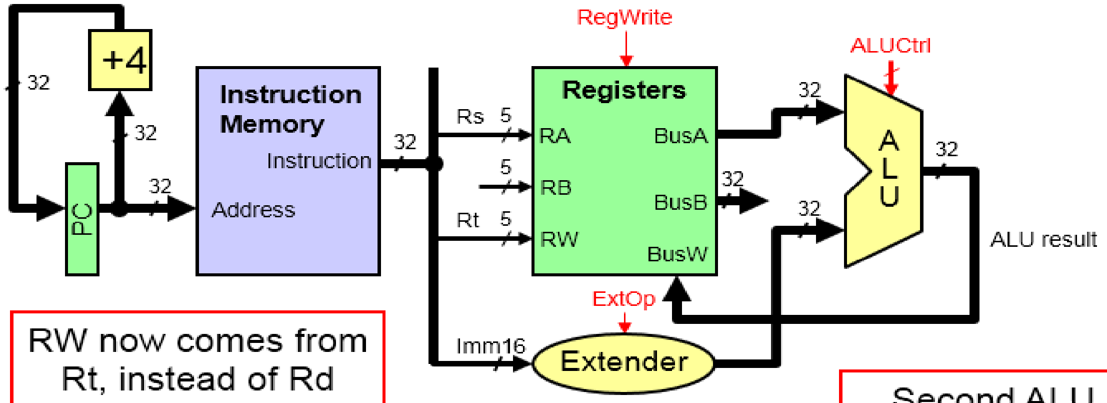
RW comes from the Rd field

ALU result is connected to BusW

- Control signals
  - ALUCtrl is derived from the funct field because Op = 0 for R-type
  - RegWrite is used to enable the writing of the ALU result

Datapath for I-type ALU Instructions

Op^6	Rs^5	Rt^5	immediate^{16}



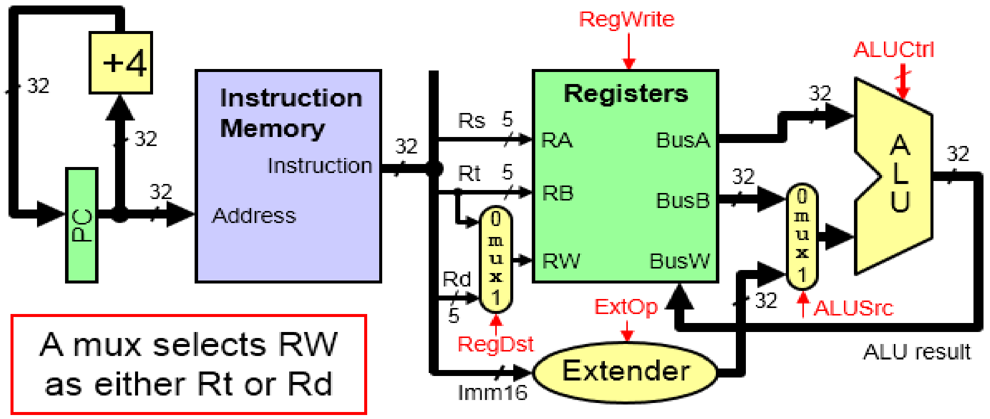
RW now comes from Rt, instead of Rd

Second ALU input comes from the extended immediate

RB and BusB are not used

- Control signals
  - ALUCtrl is derived from the Op field
  - RegWrite is used to enable the writing of the ALU result
  - ExtOp is used to control the extension of the 16-bit immediate

Combining R-type & I-type Datapaths

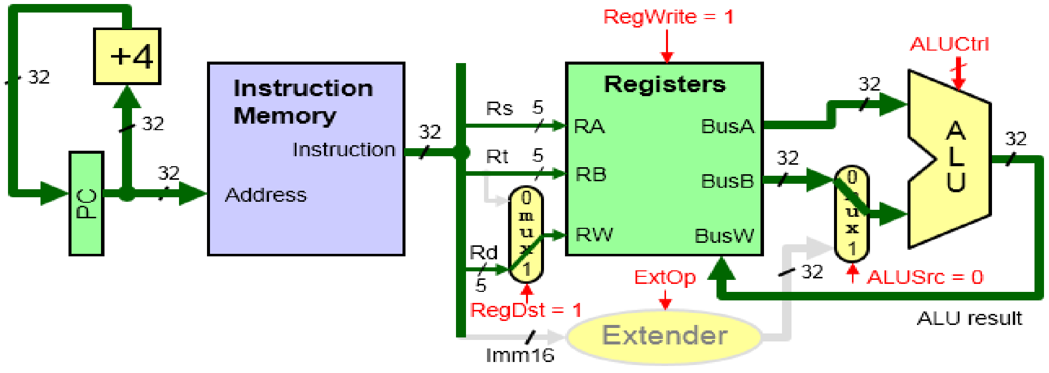


A mux selects RW as either Rt or Rd

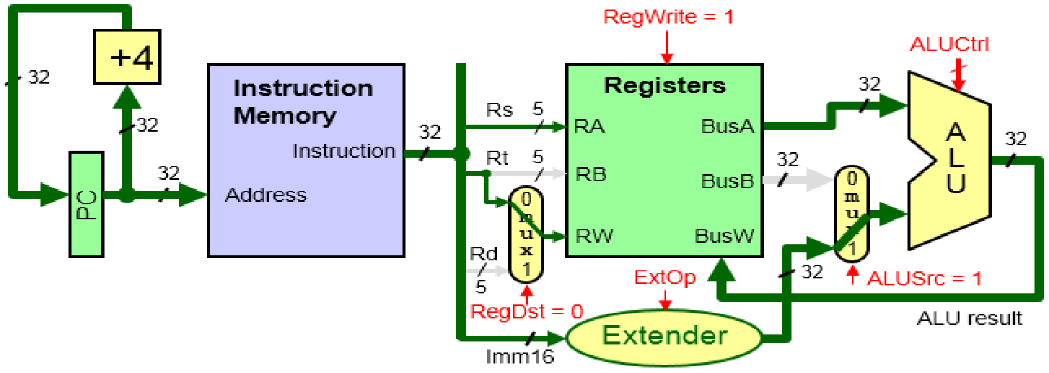
Another mux selects 2nd ALU input as either source register Rt data on BusB or the extended immediate

- ❖ Control signals
  - ✧ ALUCtrl is derived from either the Op or the funct field
  - ✧ RegWrite enables the writing of the ALU result
  - ✧ ExtOp controls the extension of the 16-bit immediate
  - ✧ RegDst selects the register destination as either Rt or Rd
  - ✧ ALUSrc selects the 2nd ALU source as BusB or extended immediate

Controlling ALU Instructions



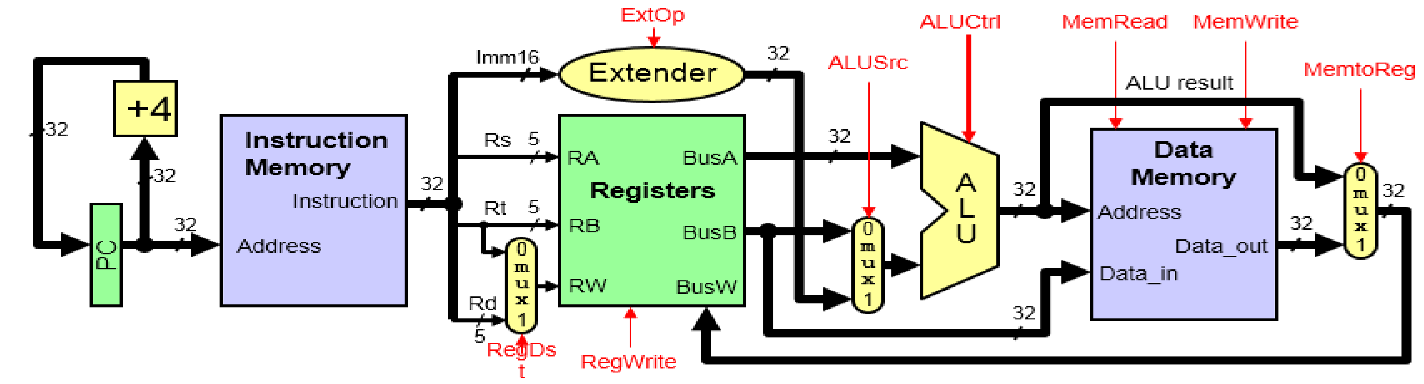
For R-type ALU instructions, RegDst is '1' to select Rd on RW and ALUSrc is '0' to select BusB as second ALU input. The active part of datapath is shown in green



For I-type ALU instructions, RegDst is '0' to select Rt on RW and ALUSrc is '1' to select Extended immediate as second ALU input. The active part of datapath is shown in green

Adding Data Memory to Datapath

- A data memory is added for load and store instructions



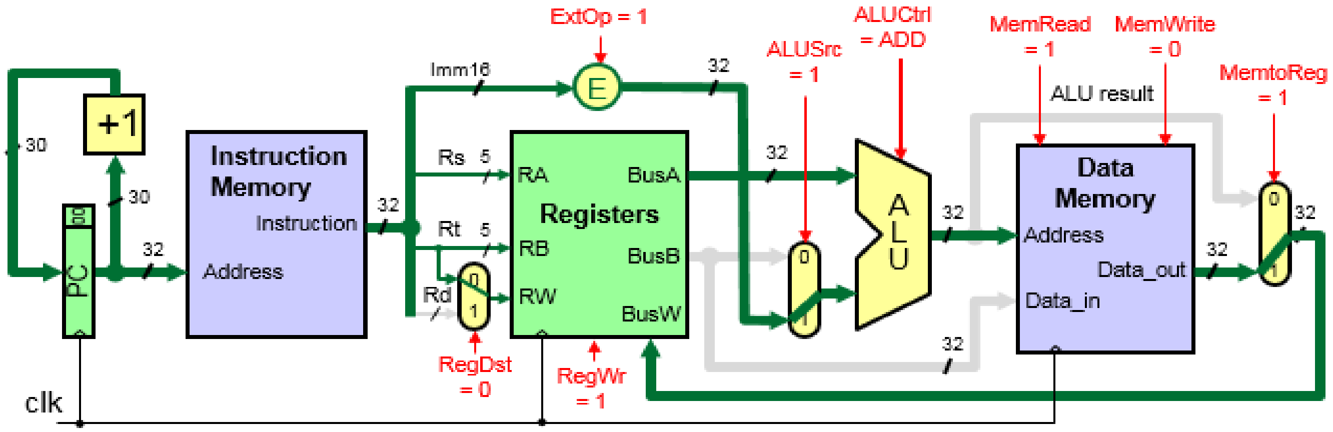
ALU calculates data memory address

A 3rd mux selects data on BusW as either ALU result or memory data_out

BusB is connected to Data_in of Data Memory for store instructions

- Additional Control signals
  - MemRead for load instructions
  - MemWrite for store instructions
  - MemtoReg selects data on BusW as ALU result or Memory Data_out

Controlling the Execution of Load



RegDst = '0' selects Rt as destination register

RegWrite = '1' to enable writing of register file

ExtOp = 1 to sign-extend Immmediate16 to 32 bits

ALUSrc = '1' selects extended immediate as second ALU input

ALUCtrl = 'ADD' to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

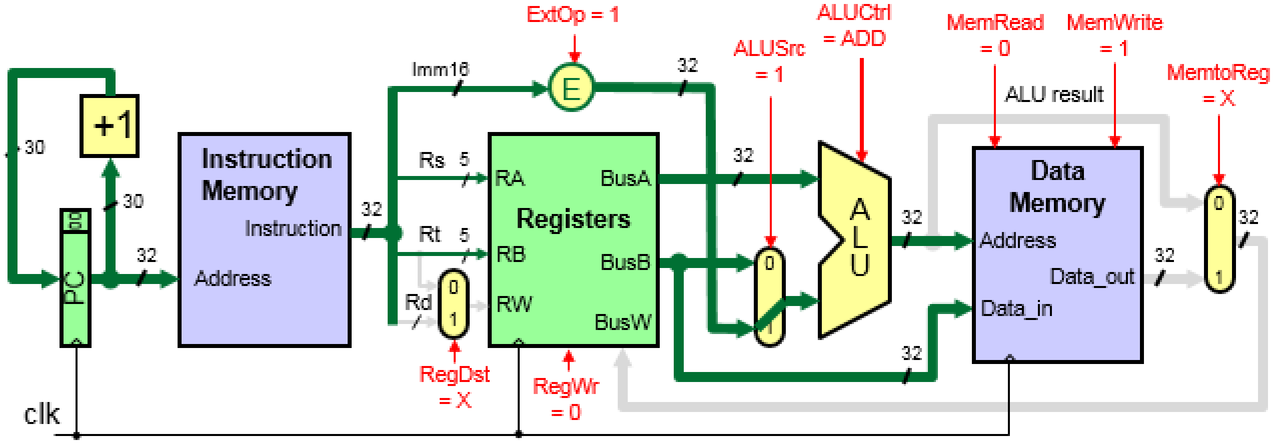
MemRead = '1' to read data memory

MemtoReg = '1' places the data read from memory on BusW

Clock edge updates PC and Register Rt

sw rt, $im^{16}(rs)$ | store word | 0x2b | rs^5 | rt^5 | im^{16}

Controlling the Execution of Store



RegDst = 'X' because no register is written

RegWrite = '0' to disable writing of register file

ExtOp = 1 to sign-extend Immmediate16 to 32 bits

ALUSrc = '1' selects extended immediate as second ALU input

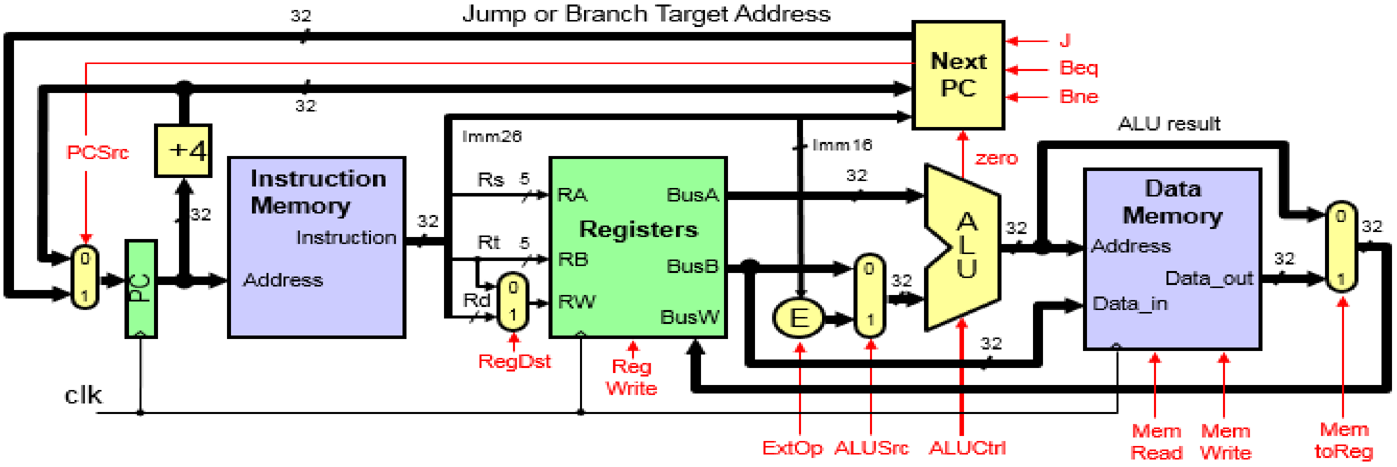
ALUCtrl = 'ADD' to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

MemWrite = '1' to write data memory

MemtoReg = 'X' because don't care what data is put on BusW

Clock edge updates PC and Data Memory

Adding Jump and Branch to Datapath

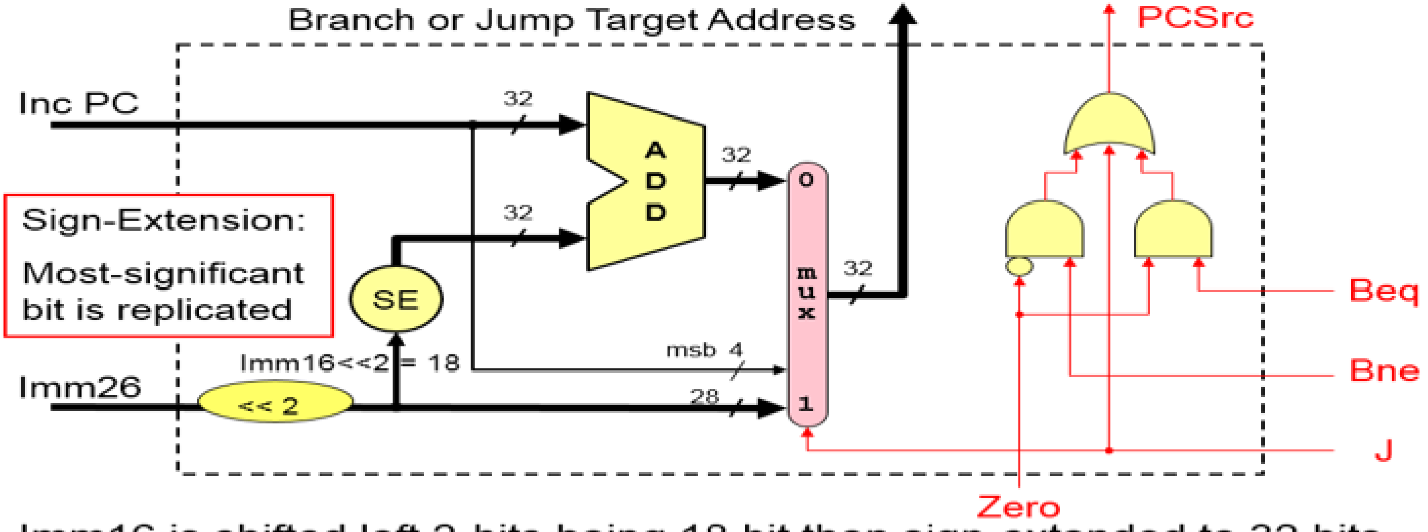


❖ Additional Control Signals

- J, Beq, Bne for jump and branch instructions
- Zero flag of the ALU is examined
- PCSrc = 1 for jump & taken branch

Next PC logic computes jump or branch target instruction address

Details of Next PC



Sign-Extension:

Most-significant bit is replicated

Imm16 is shifted left 2-bits being 18 bit then sign-extended to 32 bits

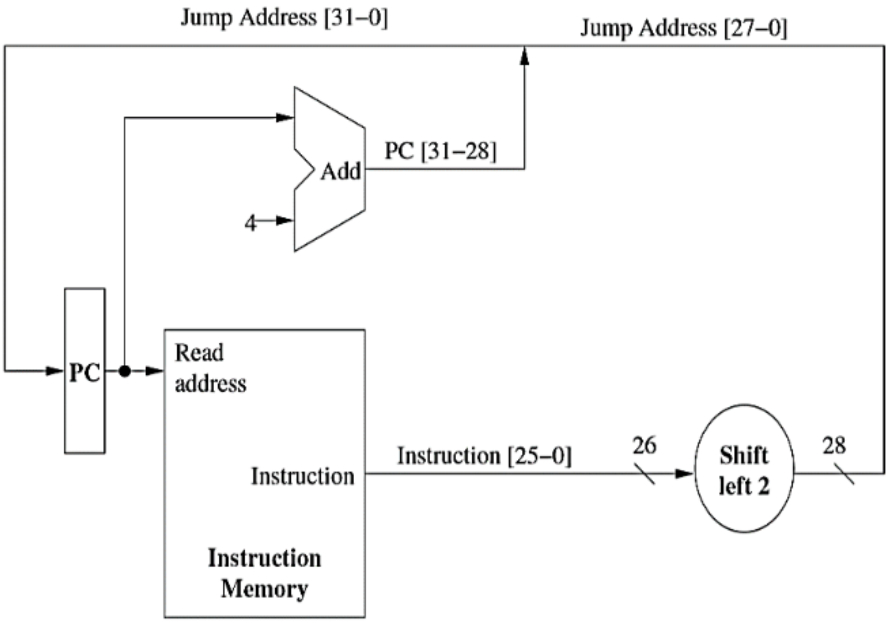
Jump target address: upper 4 bits of PC are concatenated with Imm26 bit after shifting by 2 to be {Most 4 bit of PC, 28bit}

PCSrc = J + (Beq . Zero) + (Bne . $\overline{\text{Zero}}$)

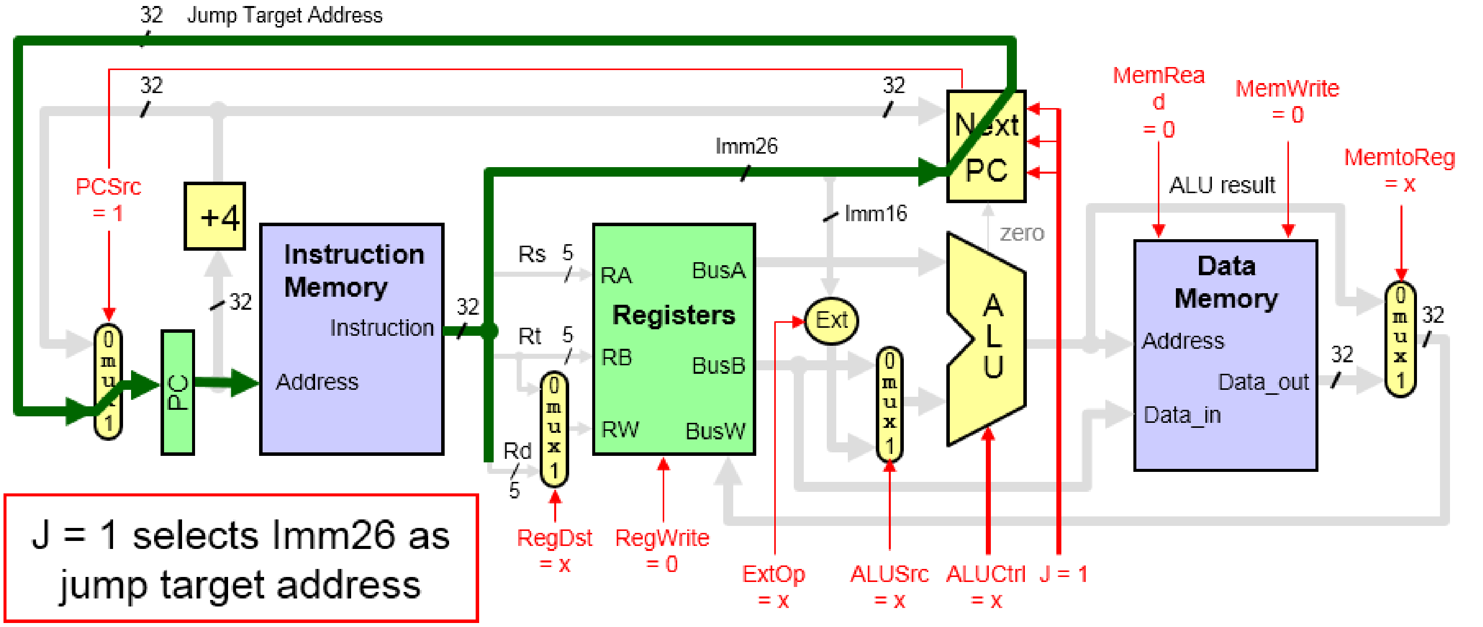
DATAPATH FOR J-FORMAT

Here, we have modified the datapath to work only for the j instruction.

```
j targaddr
```



Controlling the Execution of Jump



J = 1 selects Imm26 as jump target address

Upper 4 bits are from the incremented PC

PCSrc = 1 to select jump target address

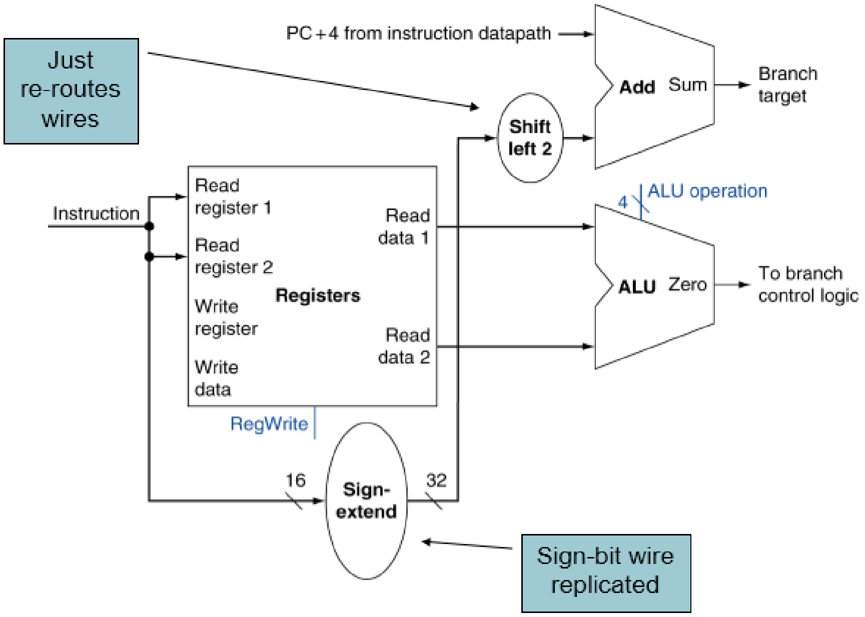
MemRead, MemWrite & RegWrite are 0

We don’t care about RegDst, ExtOp, ALUSrc, ALUCtrl, and MemtoReg

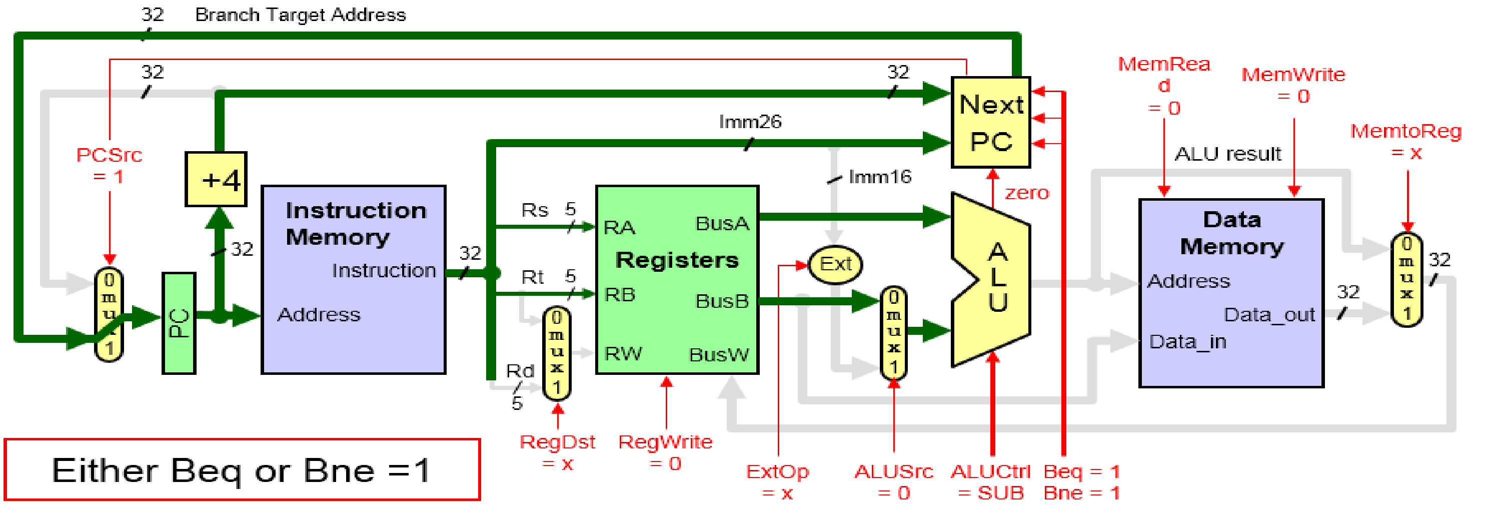
DATAPATH FOR BRANCH INSTRUCTIONS

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4

Already calculated by instruction fetch



Controlling the Execution of Branch



Either Beq or Bne =1

Next PC outputs branch target address

ALUSrc = ‘0’ (2nd ALU input is BusB)
ALUCtrl = ‘SUB’ produces zero flag

Next PC logic determines PCSrc according to zero flag

MemRead = MemWrite = RegWrite = 0

RegDst = ExtOp = MemtoReg = x

SINGLE-CYCLE CONTROL

Now we have a complete datapath for our simple MIPS subset. We will add the control.

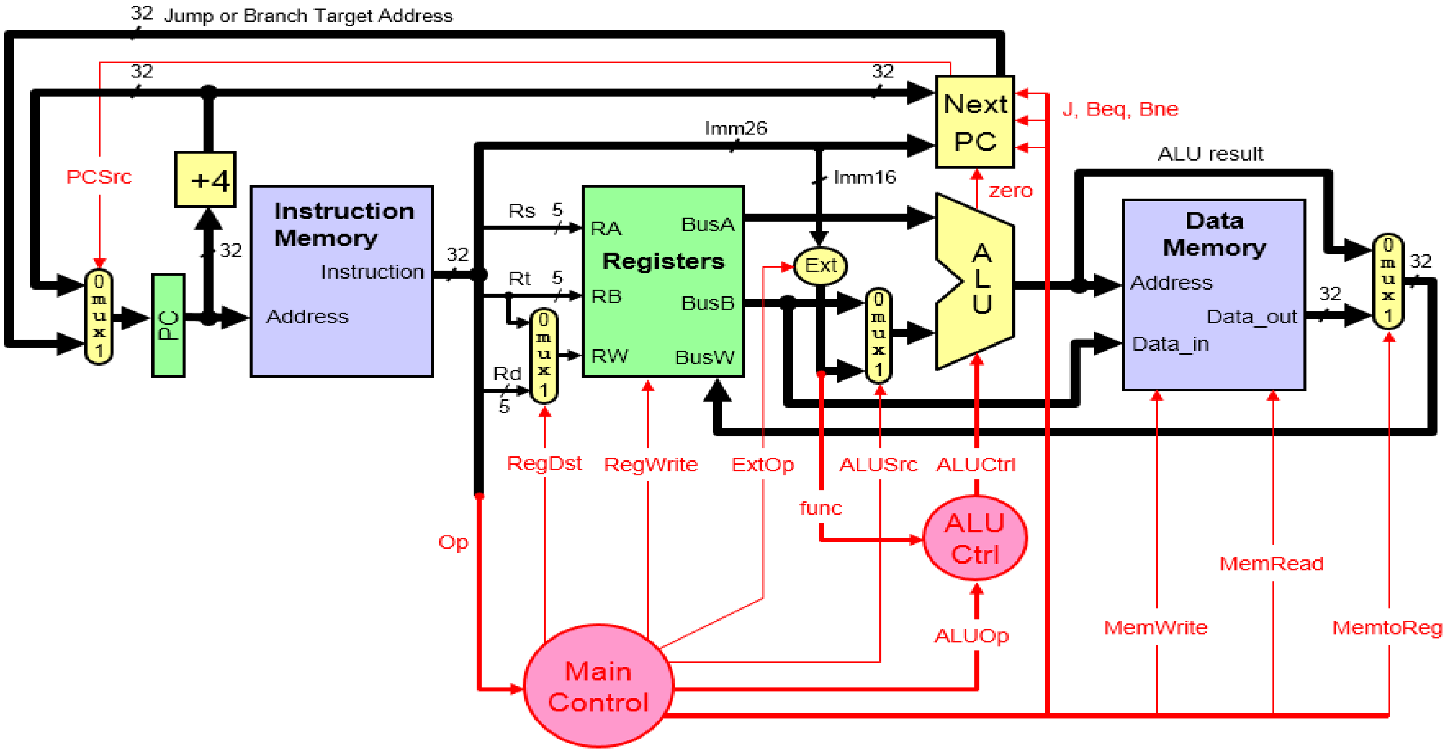
The *control unit* is responsible for taking the instruction and generating the appropriate signals for the datapath elements.

Signals that need to be generated include:-

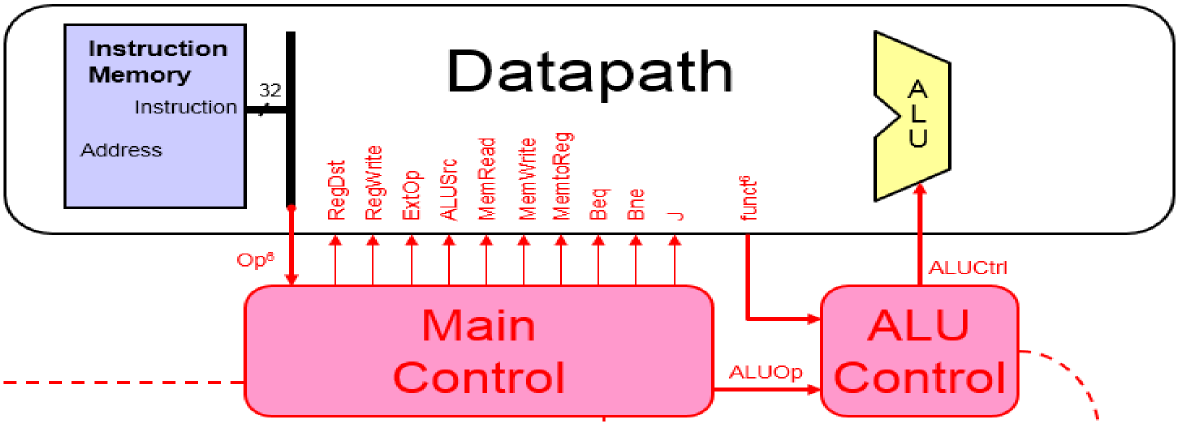
- Operation to be performed by ALU.
- Whether register file needs to be written.
- Signals for multiple intermediate multiplexors.
- Whether data memory needs to be written.

For the most part, we can generate these signals using only the opcode and *funct* fields of an instruction.

Single-Cycle Datapath + Control



Main Control and ALU Control



Input:

- ✧ 6-bit opcode field from instruction

Output:

- ✧ 10 control signals for datapath
- ✧ ALUOp for ALU Control

Input:

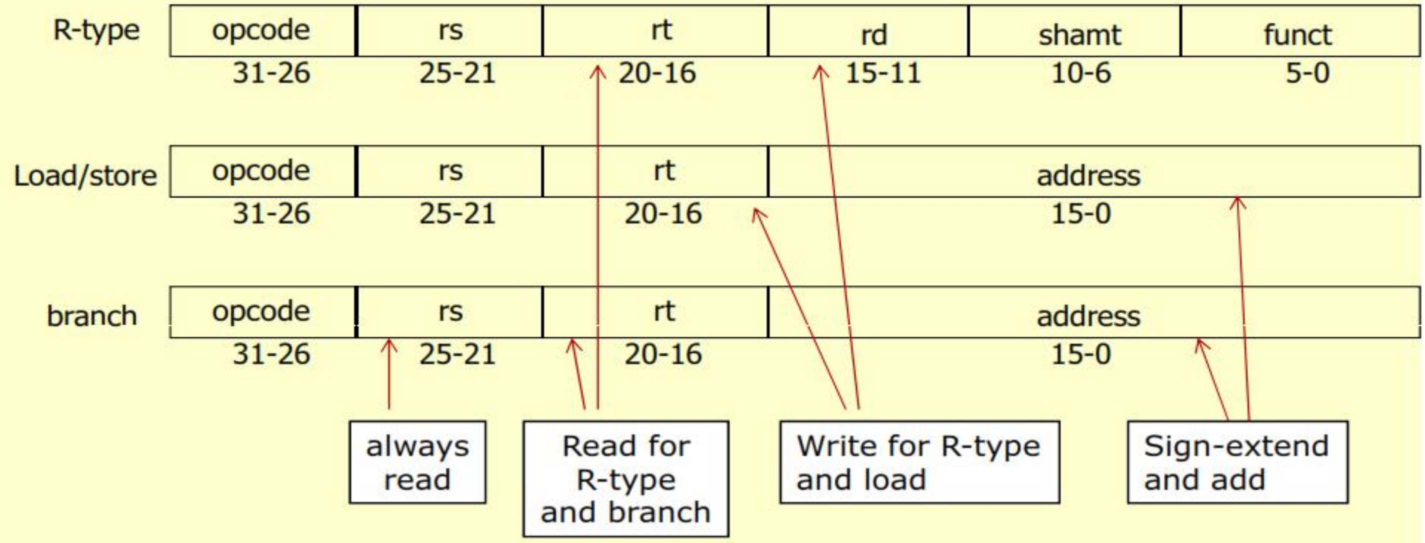
- ✧ 6-bit function field from instruction
- ✧ ALUOp from main control

Output:

- ✧ ALUCtrl signal for ALU

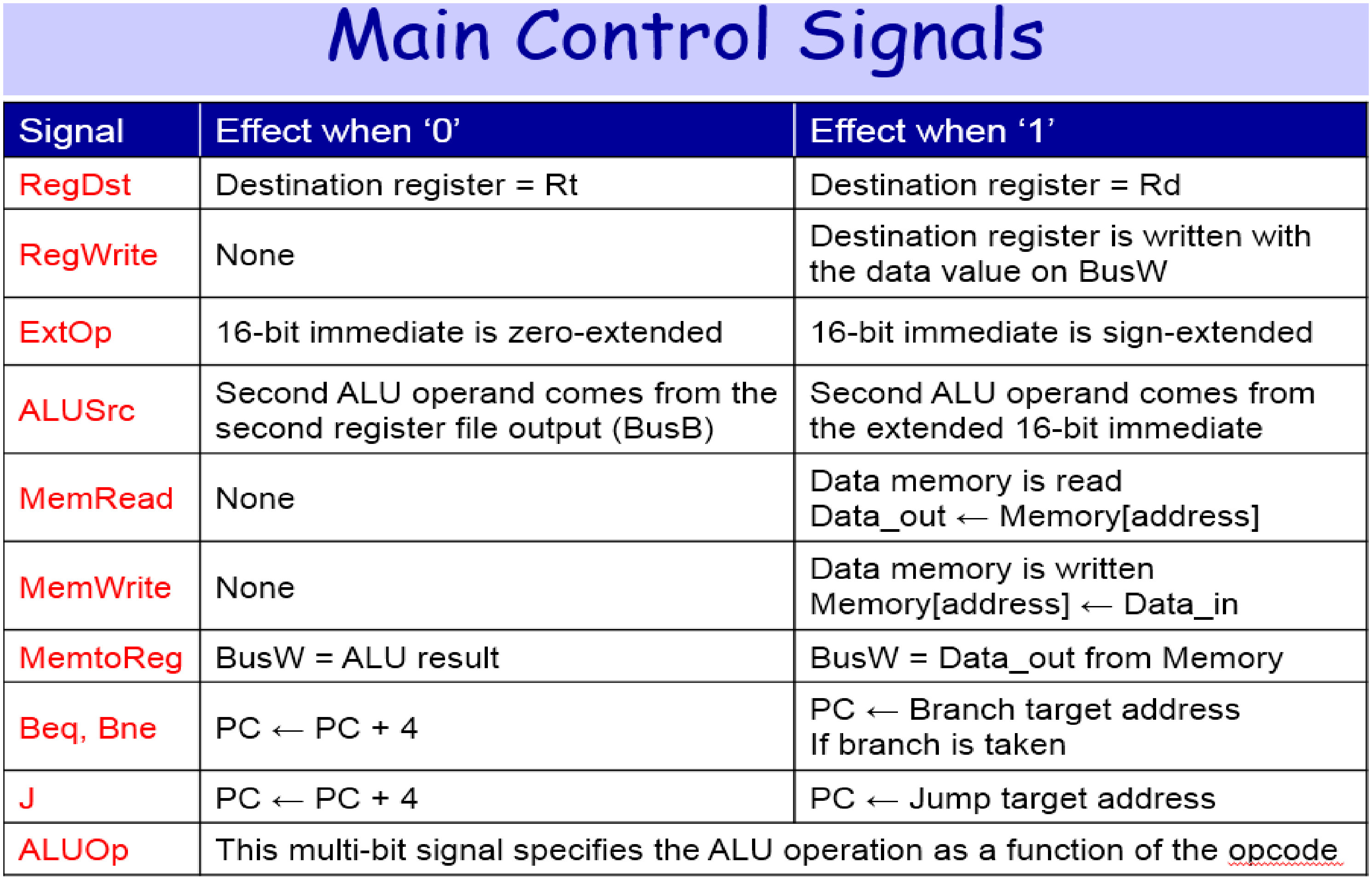
The Main Control Unit

❑ Control signals derived from the instruction



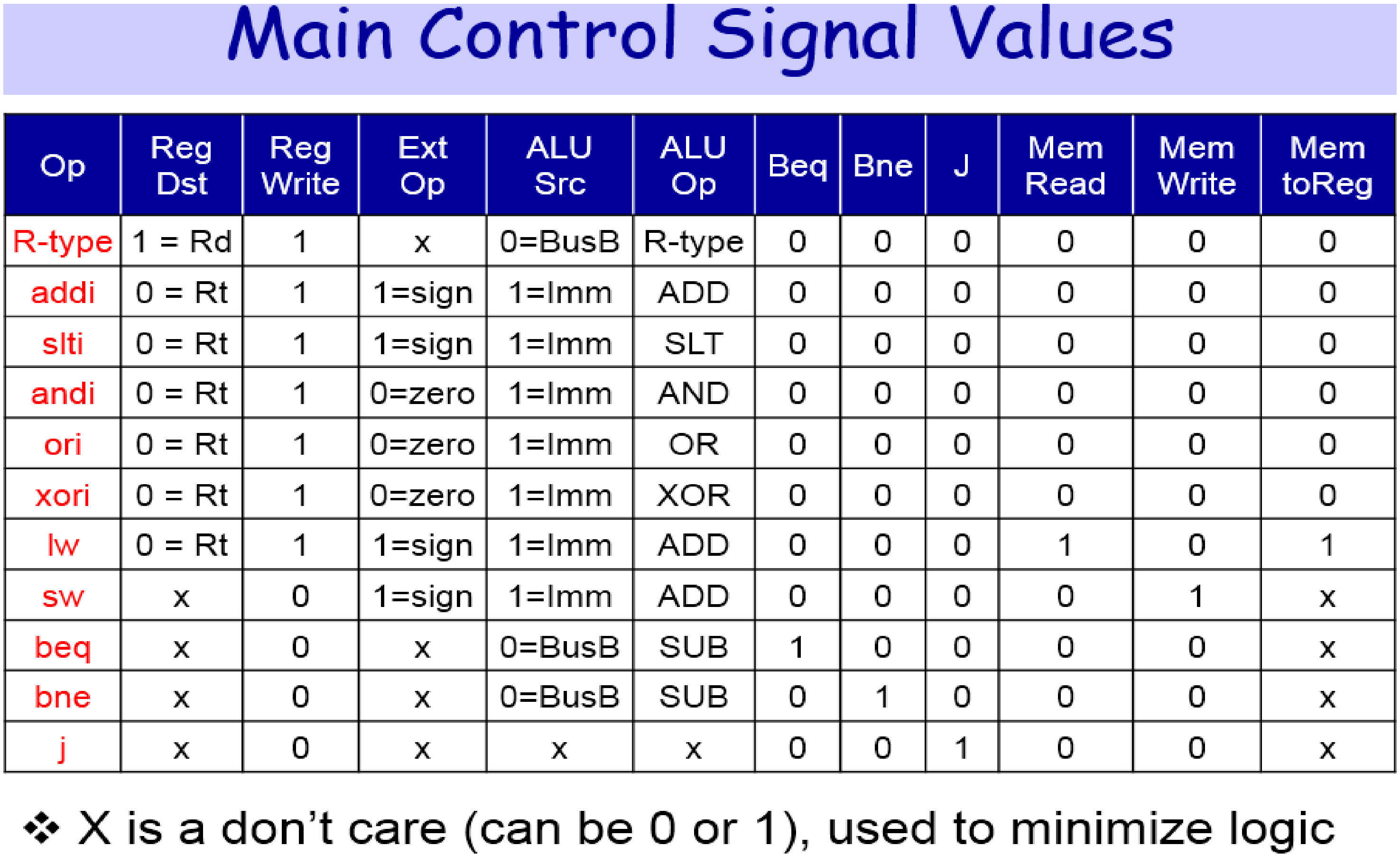
R-type	opcode	rs	rt	rd	shamt	funct
	31-26	25-21	20-16	15-11	10-6	5-0
Load/store	opcode	rs	rt	address		
	31-26	25-21	20-16	15-0		
branch	opcode	rs	rt	address		
	31-26	25-21	20-16	15-0		

❑ destination register for ***load*** instruction is in bits 20-16 (rt) while for ***R-type*** is in bits 15-11 (rd) (***will require multiplexor to select***)



Main Control Signals

Signal	Effect when '0'	Effect when '1'
RegDst	Destination register = Rt	Destination register = Rd
RegWrite	None	Destination register is written with the data value on BusW
ExtOp	16-bit immediate is zero-extended	16-bit immediate is sign-extended
ALUSrc	Second ALU operand comes from the second register file output (BusB)	Second ALU operand comes from the extended 16-bit immediate
MemRead	None	Data memory is read Data_out ← Memory[address]
MemWrite	None	Data memory is written Memory[address] ← Data_in
MemtoReg	BusW = ALU result	BusW = Data_out from Memory
Beq, Bne	PC ← PC + 4	PC ← Branch target address If branch is taken
J	PC ← PC + 4	PC ← Jump target address
ALUOp	This multi-bit signal specifies the ALU operation as a function of the opcode	



Main Control Signal Values

Op	Reg Dst	Reg Write	Ext Op	ALU Src	ALU Op	Beq	Bne	J	Mem Read	Mem Write	Mem toReg
R-type	1 = Rd	1	x	0=BusB	R-type	0	0	0	0	0	0
addi	0 = Rt	1	1=sign	1=Imm	ADD	0	0	0	0	0	0
slti	0 = Rt	1	1=sign	1=Imm	SLT	0	0	0	0	0	0
andi	0 = Rt	1	0=zero	1=Imm	AND	0	0	0	0	0	0
ori	0 = Rt	1	0=zero	1=Imm	OR	0	0	0	0	0	0
xori	0 = Rt	1	0=zero	1=Imm	XOR	0	0	0	0	0	0
lw	0 = Rt	1	1=sign	1=Imm	ADD	0	0	0	1	0	1
sw	x	0	1=sign	1=Imm	ADD	0	0	0	0	1	x
beq	x	0	x	0=BusB	SUB	1	0	0	0	0	x
bne	x	0	x	0=BusB	SUB	0	1	0	0	0	x
j	x	0	x	x	x	0	0	1	0	0	x

❖ X is a don't care (can be 0 or 1), used to minimize logic

Logic Equations for Control Signals

RegDst <= R-type

RegWrite <= $(\overline{\text{sw + beq + bne + j}})$

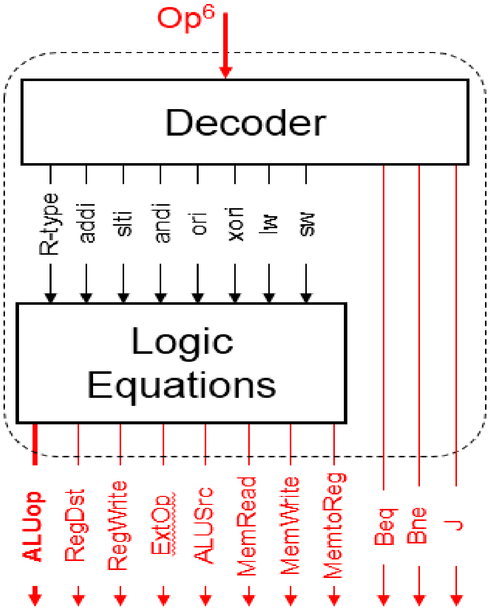
ExtOp <= $(\overline{\text{andi + ori + xori}})$

ALUSrc <= $(\overline{\text{R-type + beq + bne}})$

MemRead <= lw

MemWrite <= sw

MemtoReg <= lw



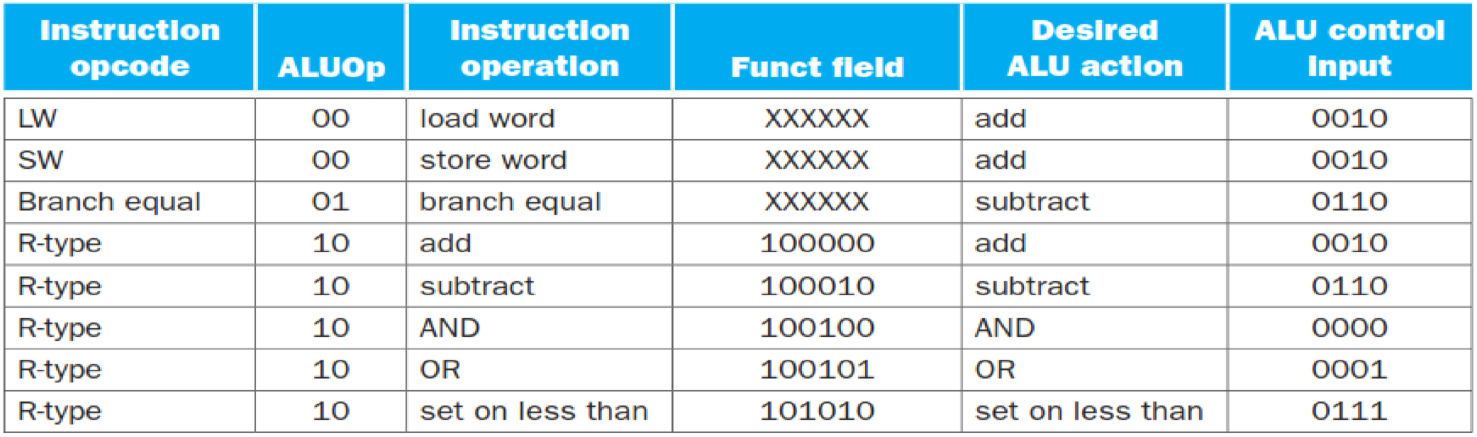
ALU Control

ALU used for

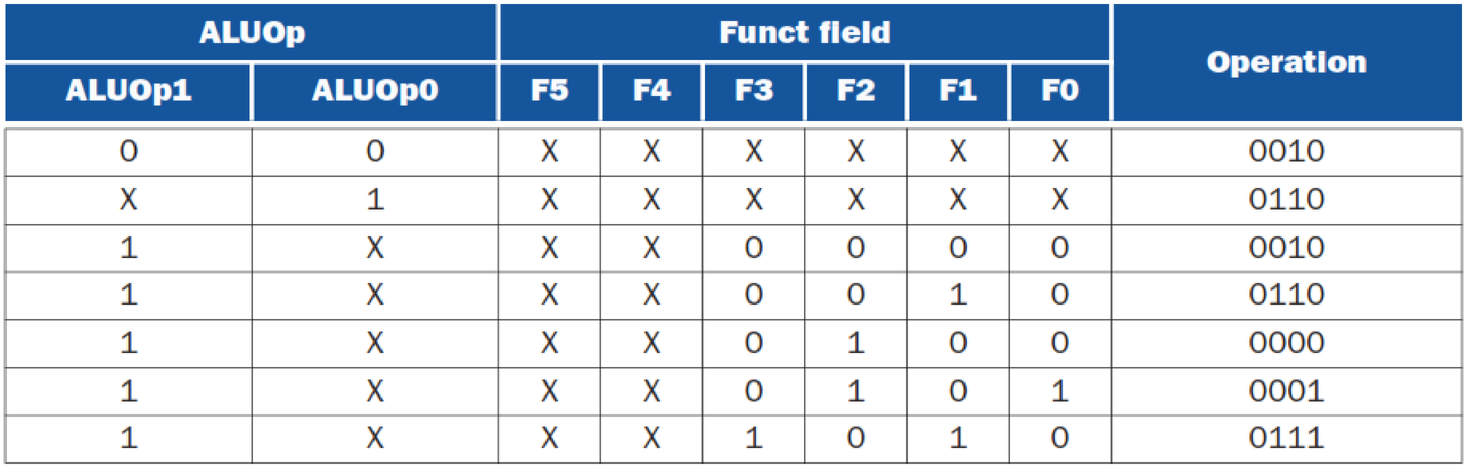
- Load/Store: **Function** = add
- Branch: **Function** = subtract
- R-type: **Function** depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than

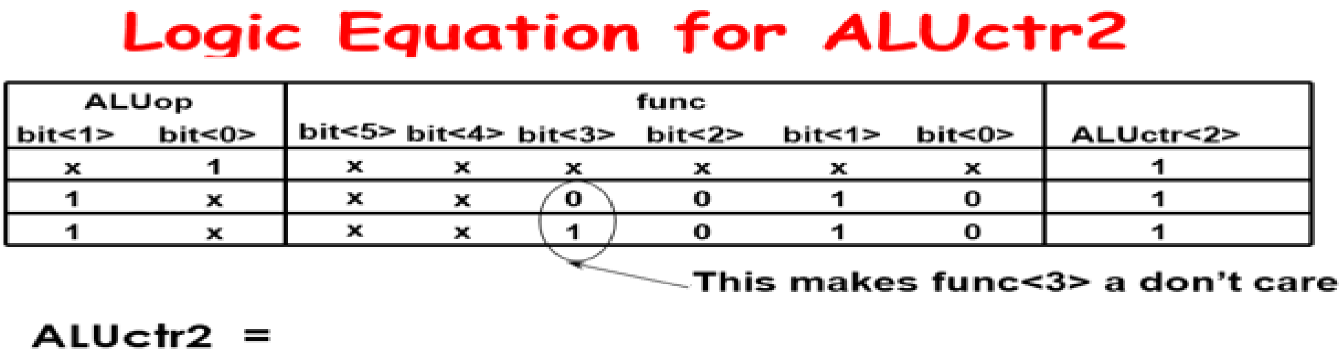
- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control



Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111



ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

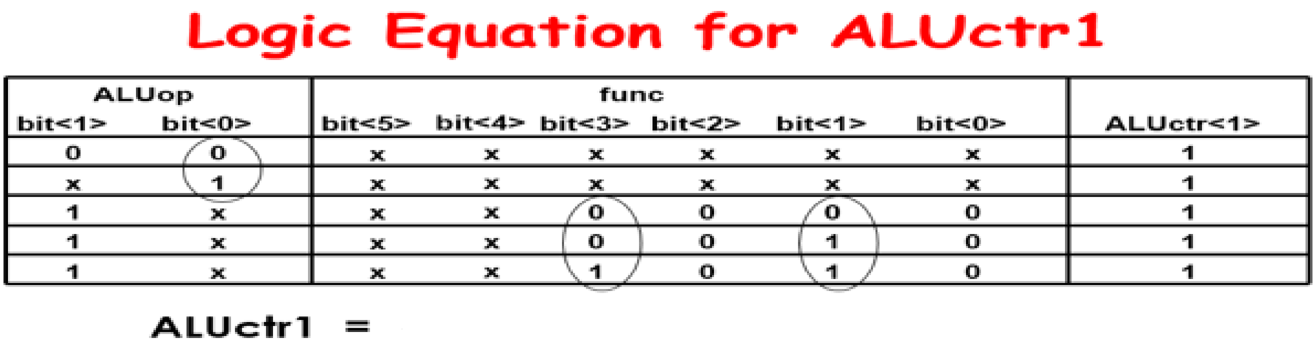


Logic Equation for ALUctr2

ALUop		func						
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<2>
x	1	x	x	x	x	x	x	1
1	x	x	x	0	0	1	0	1
1	x	x	x	1	0	1	0	1

This makes func<3> a don't care

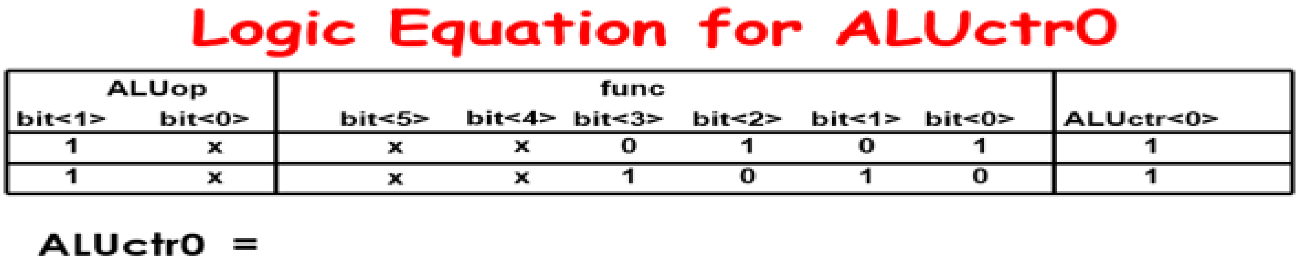
ALUctr2 =



Logic Equation for ALUctr1

ALUop		func						
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<1>
0	0	x	x	x	x	x	x	1
x	1	x	x	x	x	x	x	1
1	x	x	x	0	0	0	0	1
1	x	x	x	0	0	1	0	1
1	x	x	x	1	0	1	0	1

ALUctr1 =

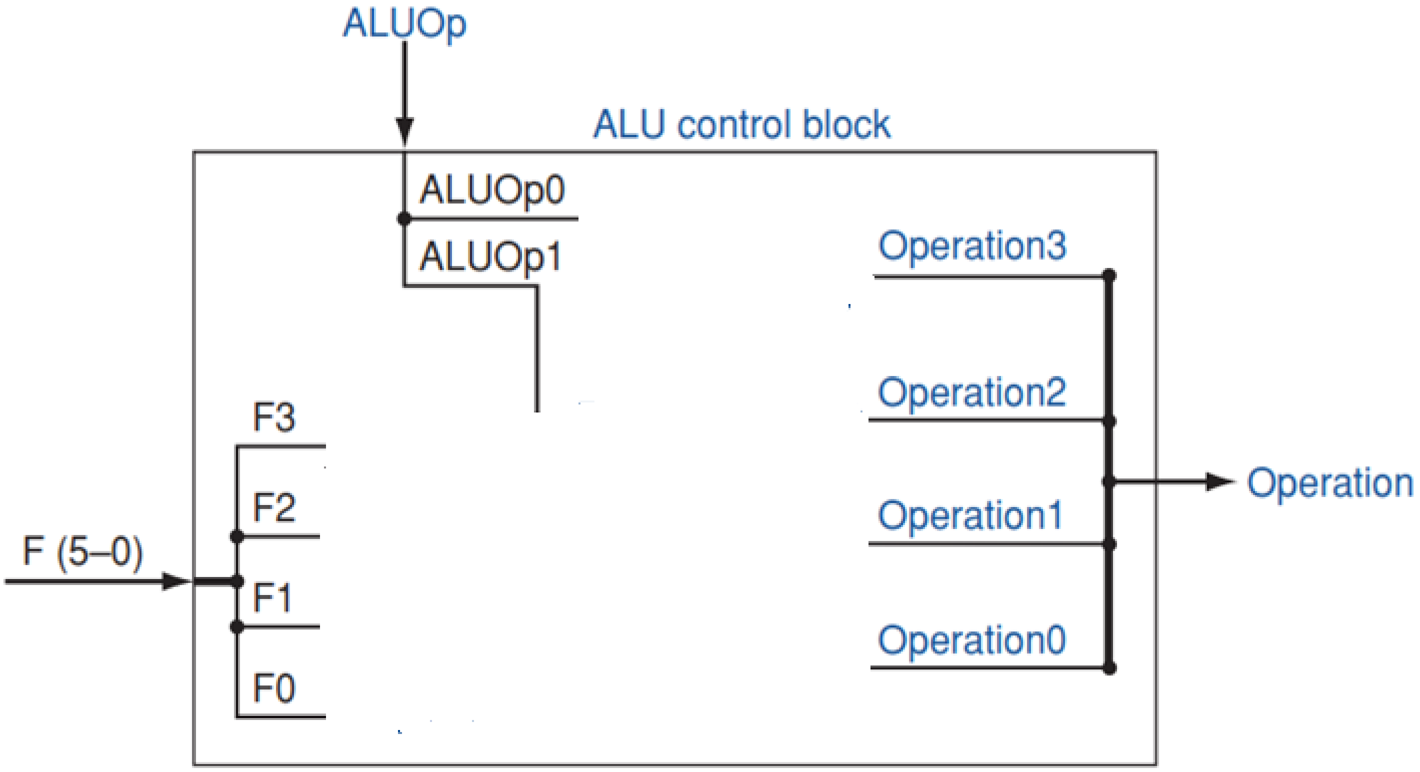


Logic Equation for ALUctr0

ALUop		func						
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
1	x	x	x	0	1	0	1	1
1	x	x	x	1	0	1	0	1

ALUctr0 =

The ALU control block generates the four ALU control bits, based on the function code and ALUOp bits



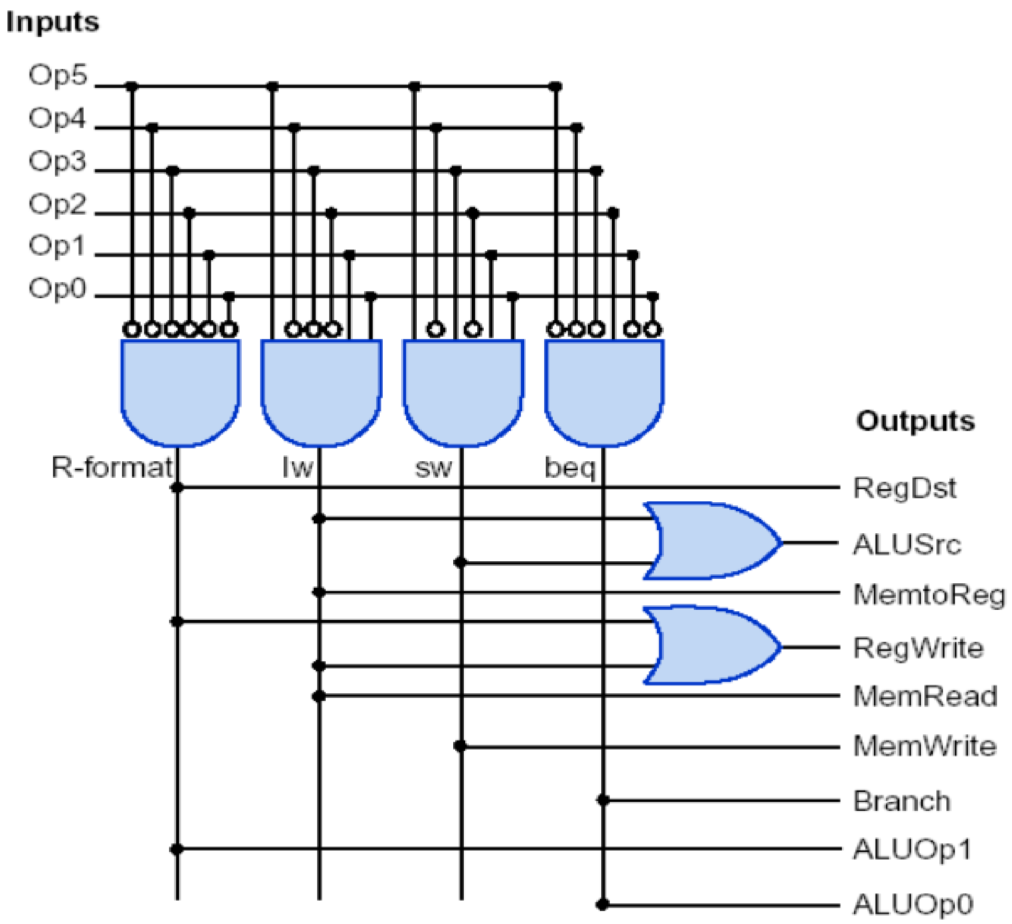
Main Control

Setting of the control signals

Instru-ction	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALU Op1	ALU Op0
R type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

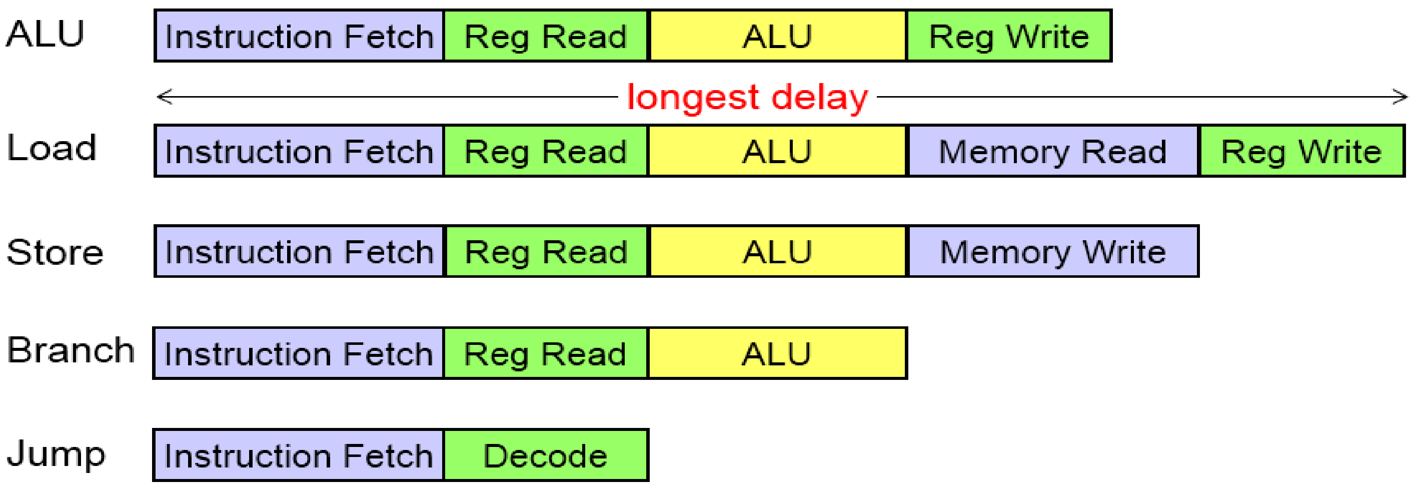
Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Control Unit PLA Implementation



Drawbacks of Single Cycle Processor

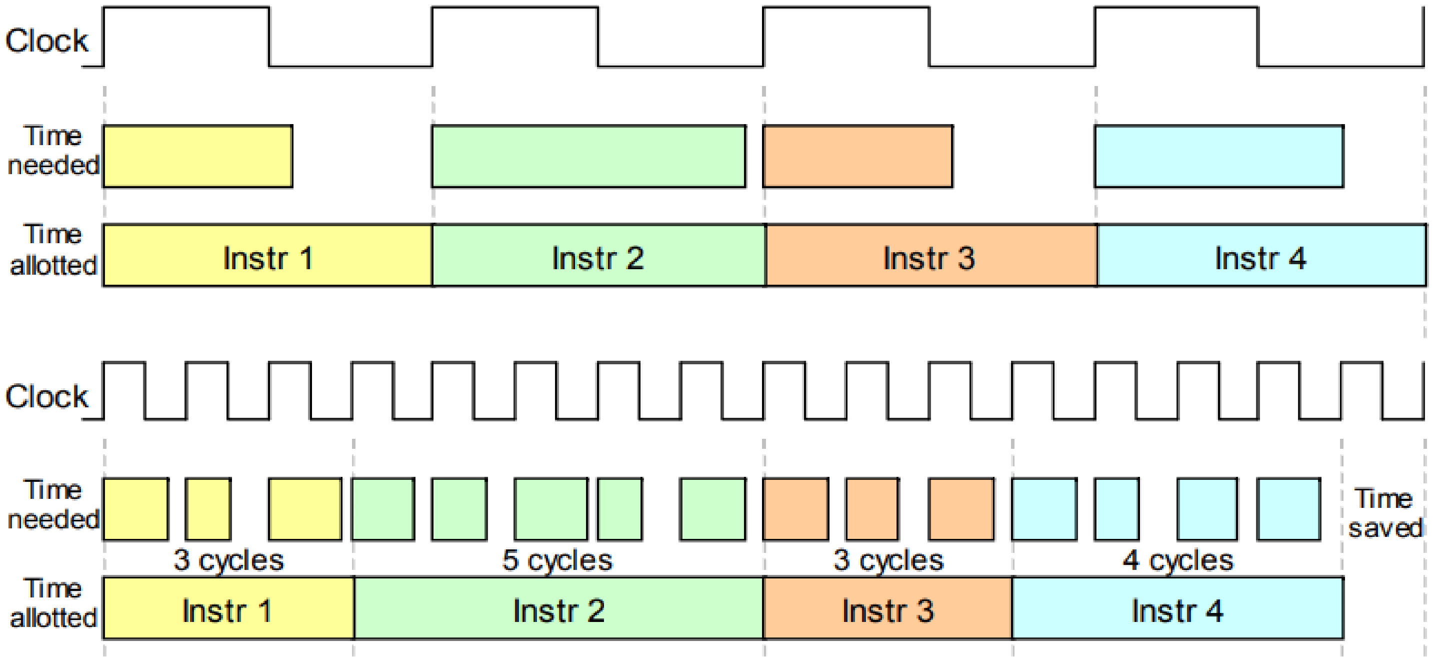
- Long cycle time
  - All instructions take as much time as the slowest



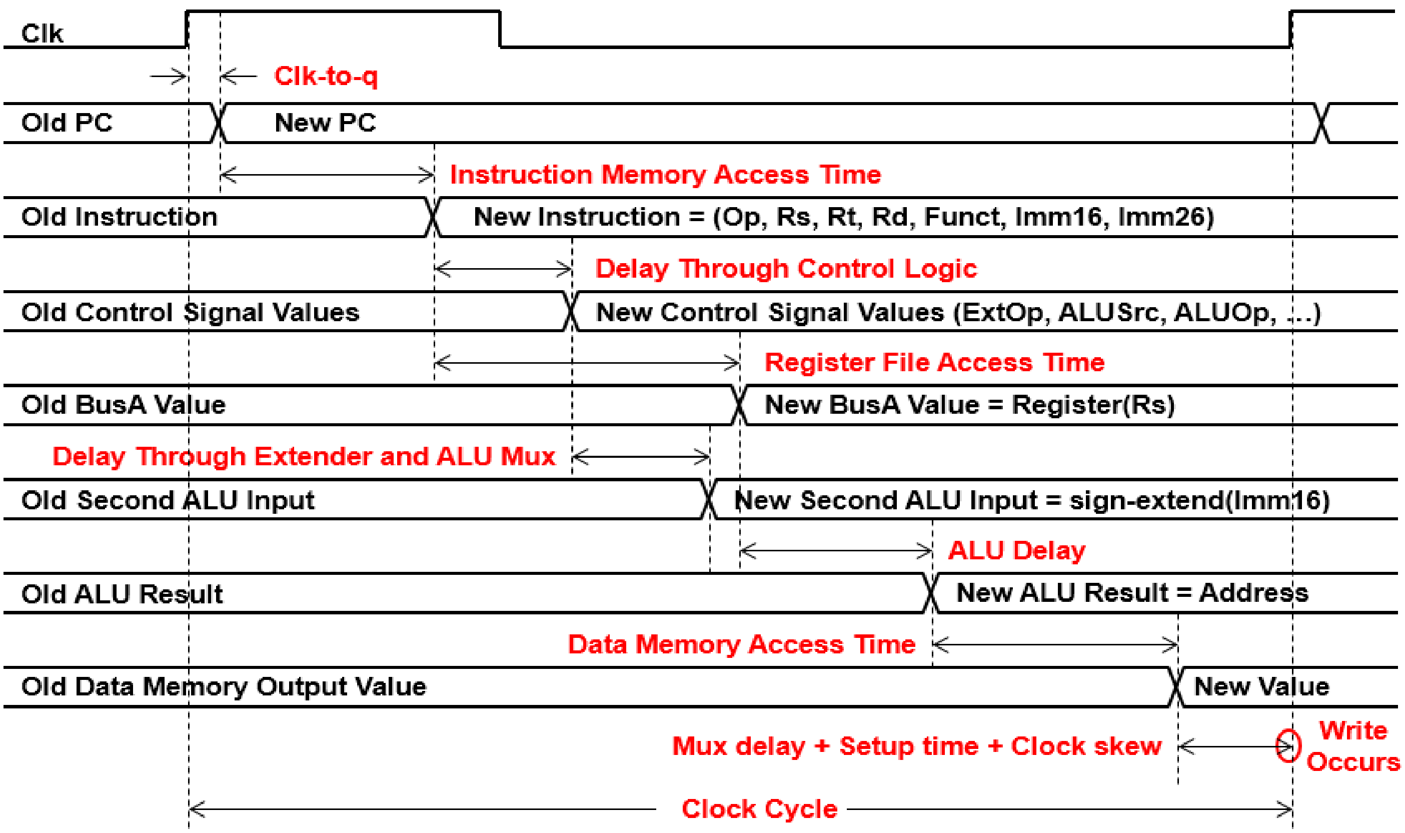
ALU	Instruction Fetch	Reg Read	ALU	Reg Write	
Load	Instruction Fetch	Reg Read	ALU	Memory Read	Reg Write
Store	Instruction Fetch	Reg Read	ALU	Memory Write	
Branch	Instruction Fetch	Reg Read	ALU		
Jump	Instruction Fetch	Decode			

- Alternative Solution: Multicycle implementation
  - Break down instruction execution into multiple cycles

Single-Cycle vs. Multicycle MicroMIPS



Worst Case Timing (Load Instruction)


Clk
Clk-to-q
Old PC
New PC
Instruction Memory Access Time
Old Instruction
New Instruction = (Op, Rs, Rt, Rd, Funct, Imm16, Imm26)
Delay Through Control Logic
Old Control Signal Values
New Control Signal Values (ExtOp, ALUSrc, ALUOp, ...)
Register File Access Time
Old BusA Value
New BusA Value = Register(Rs)
Delay Through Extender and ALU Mux
Old Second ALU Input
New Second ALU Input = sign-extend(Imm16)
ALU Delay
Old ALU Result
New ALU Result = Address
Data Memory Access Time
Old Data Memory Output Value
New Value
Mux delay + Setup time + Clock skew
Write Occurs
Clock Cycle

- Long cycle time: must be long enough for Load operation

  PC's Clk-to-Q

  + Instruction Memory's Access Time

  + Maximum of (

  Register File's Access Time,

  Delay through control logic + extender + ALU mux)

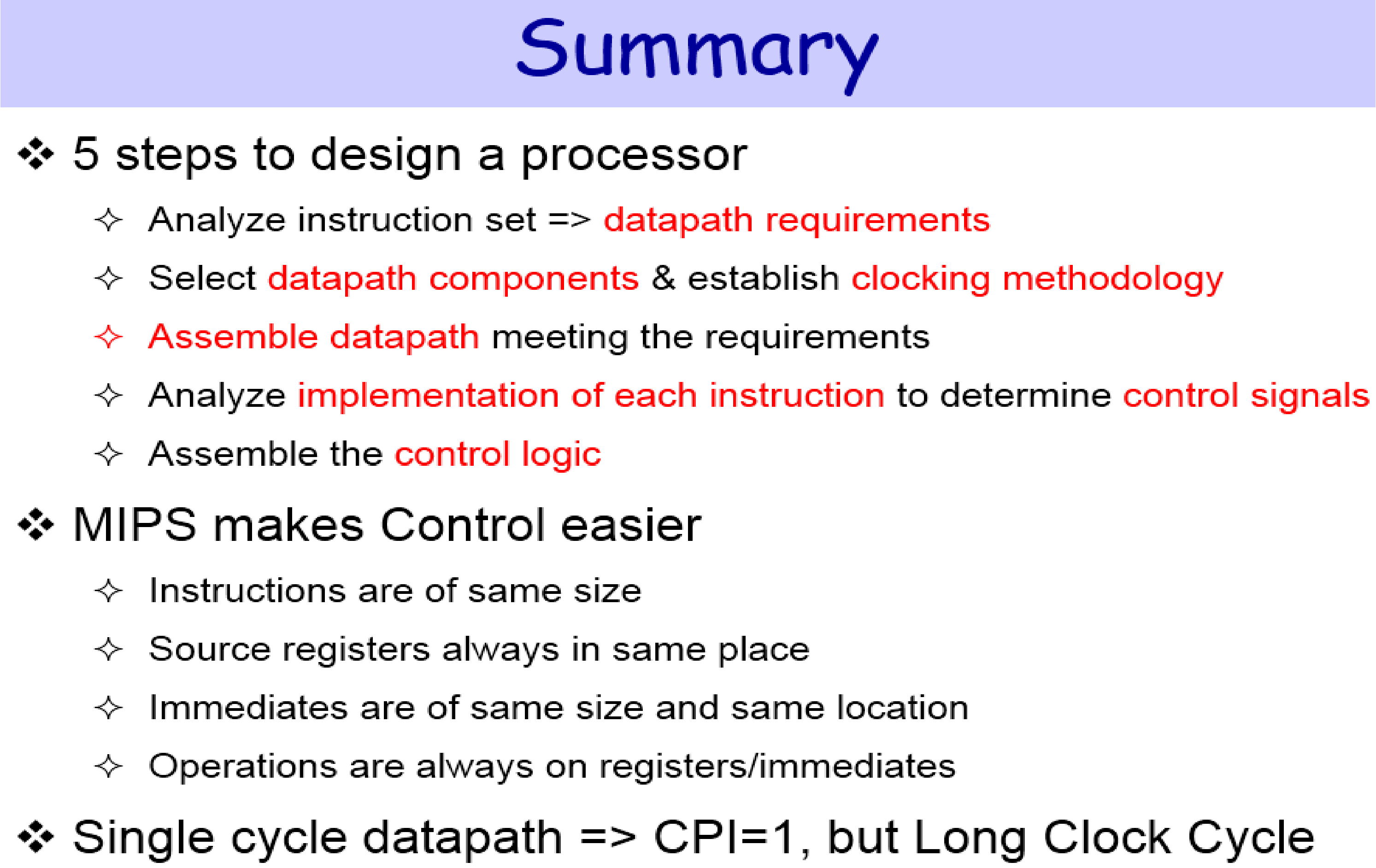
  + ALU to Perform a 32-bit Add

  + Data Memory Access Time

  + Delay through MemtoReg Mux

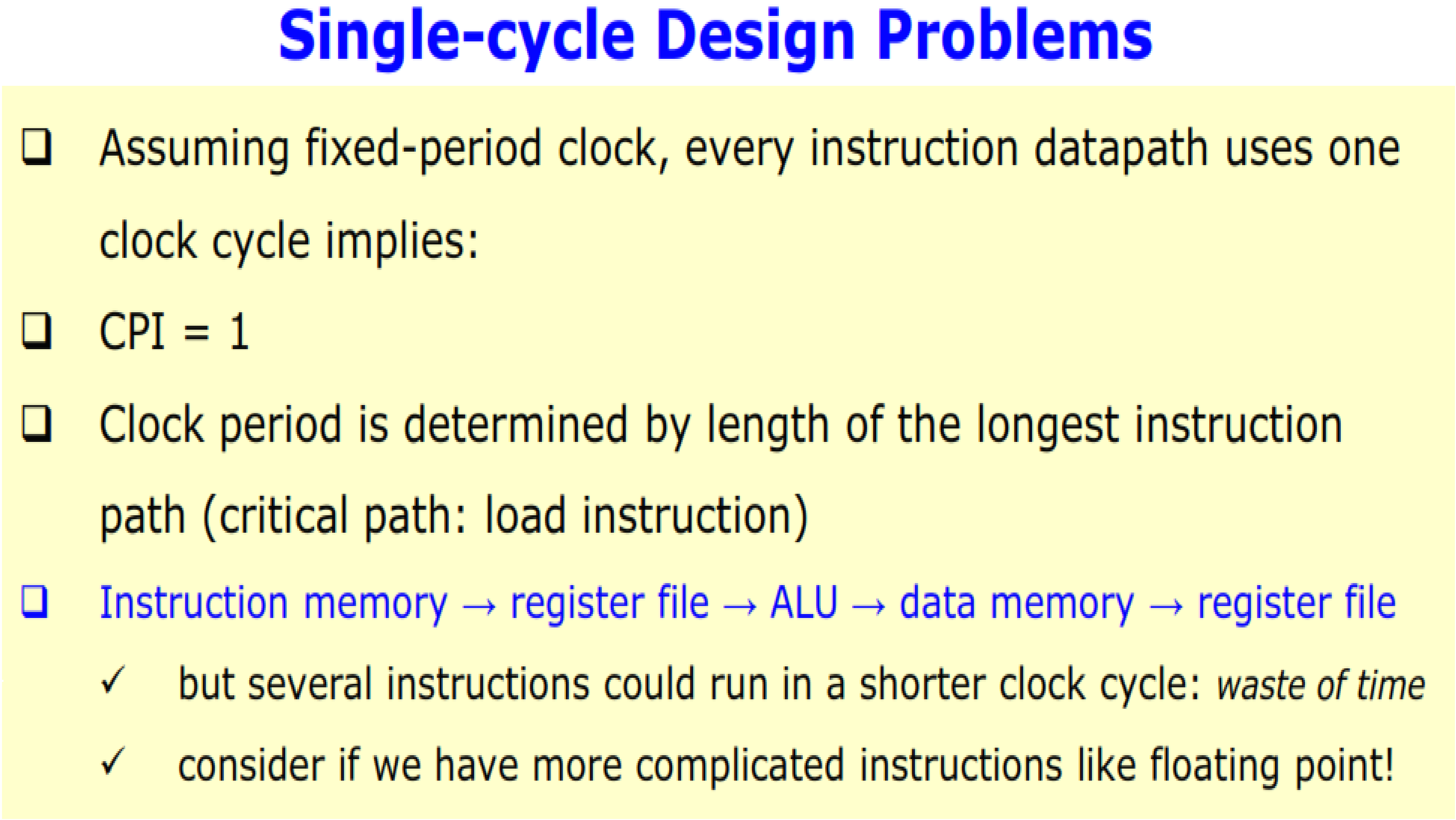
  + Setup Time for Register File Write + Clock Skew

- Cycle time is longer than needed for other instructions
  - Therefore, single cycle processor design is not used in practice



Summary

- 5 steps to design a processor
  - Analyze instruction set => datapath requirements
  - Select datapath components & establish clocking methodology
  - Assemble datapath meeting the requirements
  - Analyze implementation of each instruction to determine control signals
  - Assemble the control logic
- MIPS makes Control easier
  - Instructions are of same size
  - Source registers always in same place
  - Immediates are of same size and same location
  - Operations are always on registers/immediates
- Single cycle datapath => CPI=1, but Long Clock Cycle



Single-cycle Design Problems

- ❑ Assuming fixed-period clock, every instruction datapath uses one clock cycle implies:
- ❑ CPI = 1
- ❑ Clock period is determined by length of the longest instruction path (critical path: load instruction)
- ❑ Instruction memory → register file → ALU → data memory → register file
  - ✓ but several instructions could run in a shorter clock cycle: *waste of time*
  - ✓ consider if we have more complicated instructions like floating point!

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20

$$\begin{aligned} T_c &= t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}} \\ &= [30 + 2(250) + 150 + 200 + 25 + 20]\ \text{ps} \\ &= 925\ \text{ps} \end{aligned}$$

What's the max clock frequency?

- For a program with 100 billion instructions executing on a single-cycle MIPS processor,
  - Execution Time
    $= \text{Num. ofinstructions} \times \text{CPI} \times T_C$
    $= (100 \times 10^{9})(1)(925 \times 10^{-12}\ \text{s})$
    $= 92.5$ seconds

Fixed-period clock vs. variable-period clock

❑ **Example**

❑ Consider a machine with an additional floating point unit. Assume the functional unit delays as follows:

✓ *Mem.*: 2 ns, *ALU*: 2 ns, *FPU add*: 8 ns, *FPU multiply*: 16 ns, *register file access* (read or write): 1 ns.

✓ *multiplexors*, *control unit*, *PC accesses*, *sign extension:* no delay

❑ Assume instruction mix as follows:

✓ Loads:31%, Stores: 21%, R-type: 27%, branches: 5%, jumps 2%, FP: 7%

❑ Compare the performance of a single-cycle implementation using:

✓ *a fixed-period clock*

✓ *a variable-period clock where each instruction executes in one clock cycle that is only as long as it needs to be*

❑ **Solution**

Instruction class	Instr. mem.	Reg. read	ALU oper.	Data mem.	Reg. Write	FPU add/sub	FPU mul/div	Total time (ns)
Load word	2	1	2	2	1			8
Store word	2	1	2	2				7
R-format	2	1	2	0	1			6
Branch	2	1	2					5
Jump	2							2
FP add/sub	2	1			1	8		12
FP mul/div	2	1			1		16	20

❑ Clock period for fixed-period clock = longest instruction time = 20 ns.

❑ Average clock period for variable-period clock =

$8 \times 31\% + 7 \times 21\% + 6 \times 27\% + 5 \times 5\% + 2 \times 2\% + 20 \times 7\% + 12 \times 7\% = 7$ ns.

❑ Therefore, $\text{performance}_{\text{var-period}} / \text{performance}_{\text{fixed-period}} = 20/7 = 2.9$

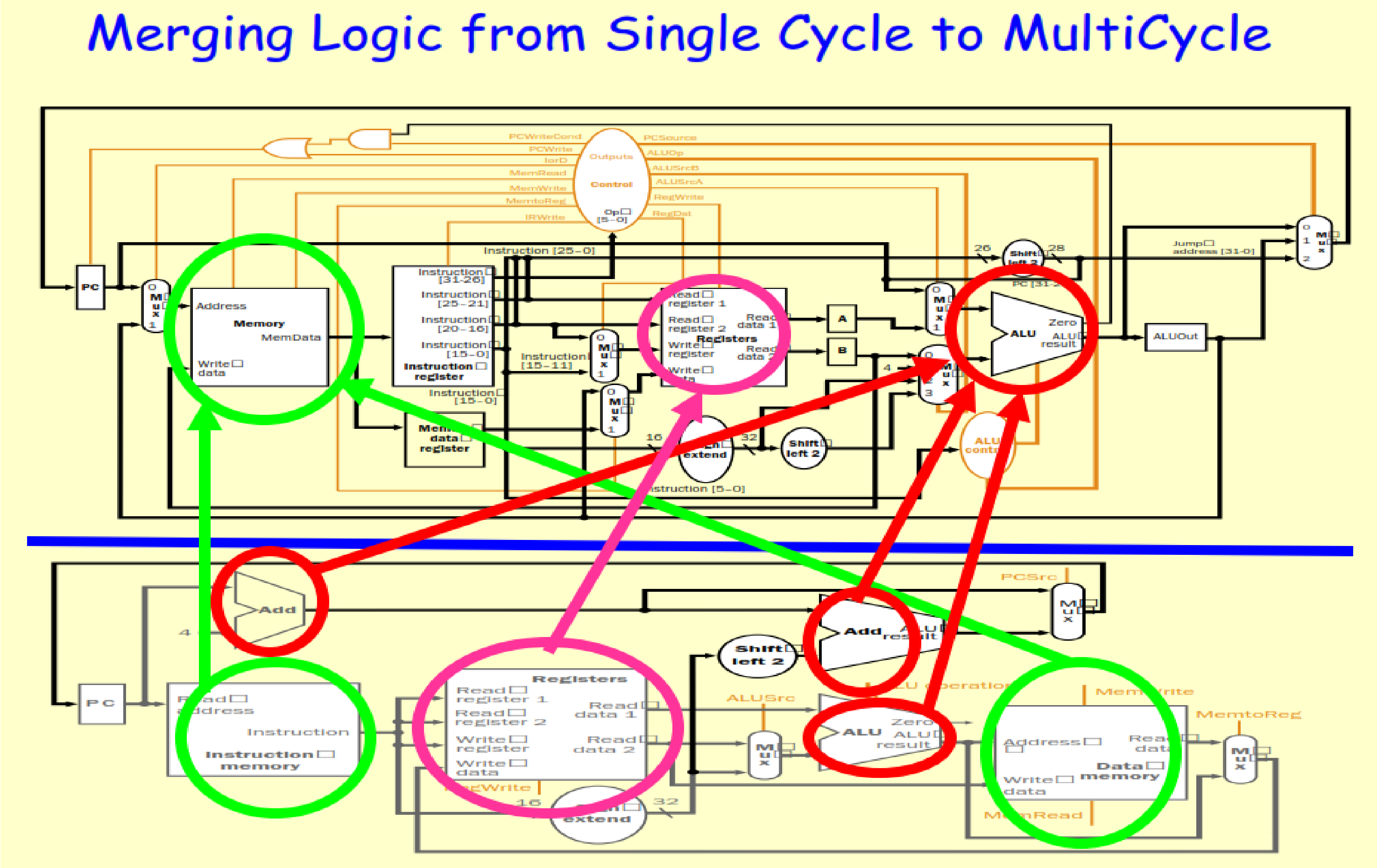
Multicycle Implementation

Multicycle implementation Also called multiple clock cycle implementation. An implementation in which an instruction is executed in multiple clock cycles.

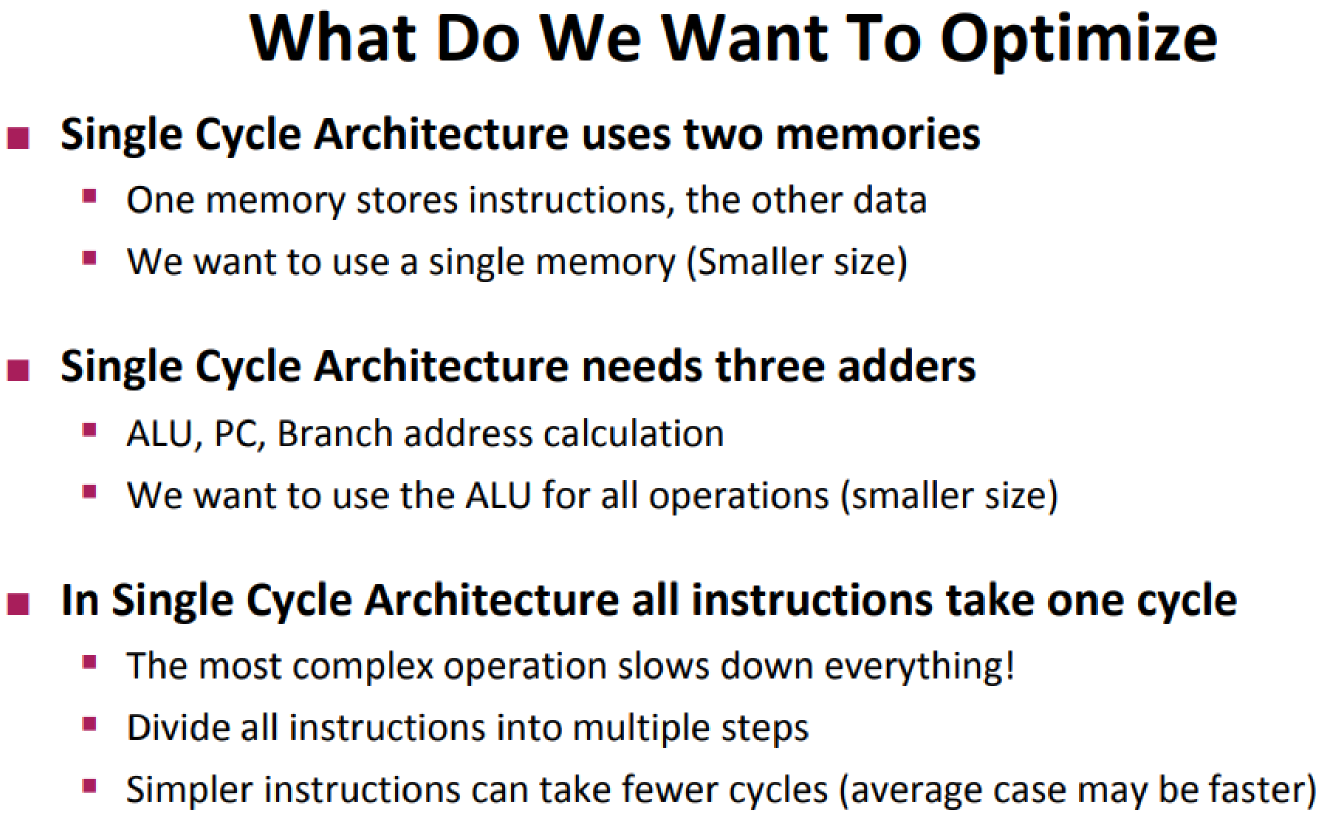
- **Single-cycle microarchitecture:**
  - + simple
  - - cycle time limited by longest instruction (lw)
  - - two adders/ALUs and two memories

- **Multi-cycle microarchitecture:**
  - + higher clock speed
  - + simpler instructions run faster
  - + reuse expensive hardware on multiple cycles

- **Same design steps: datapath & control**

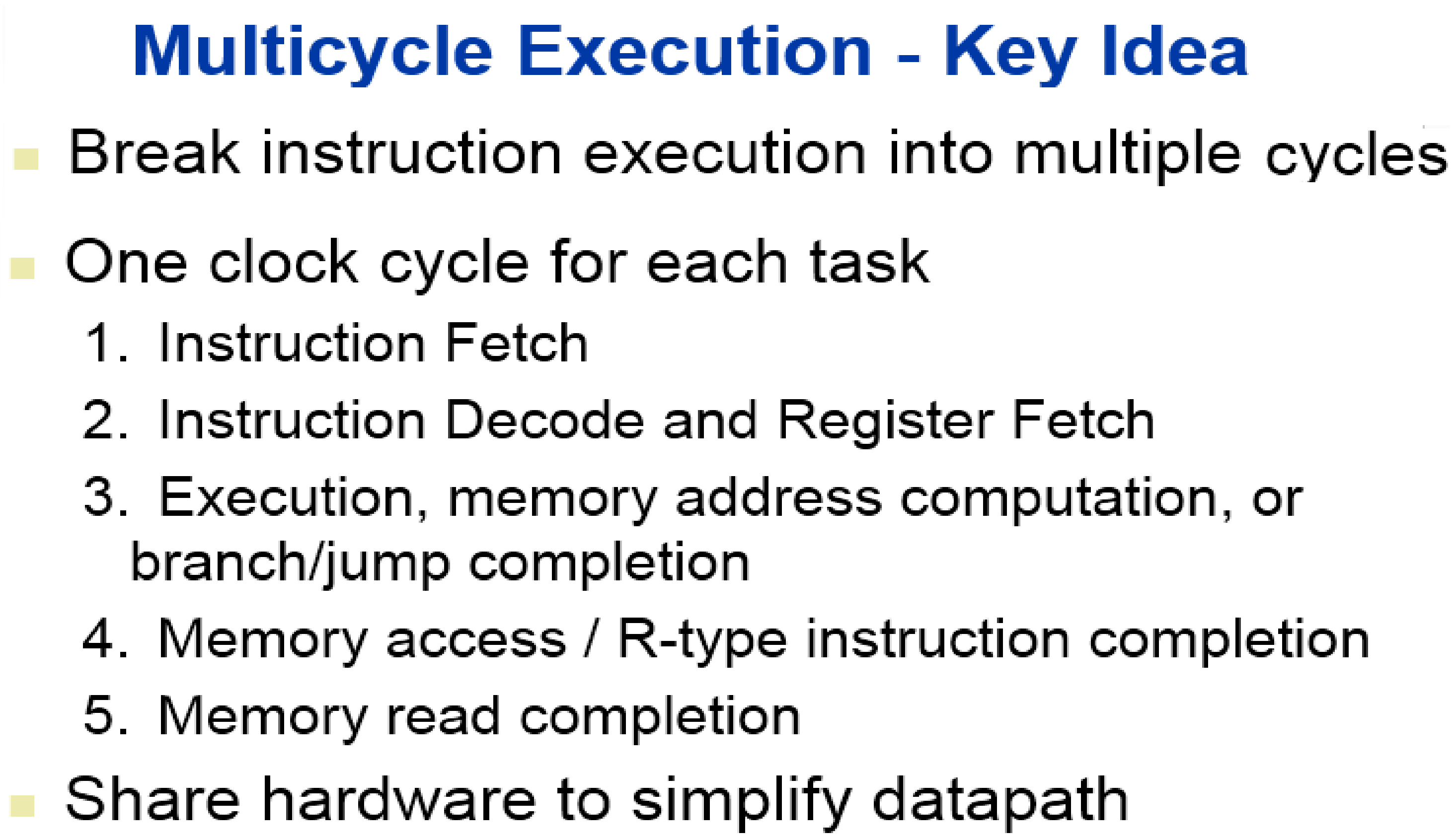


Merging Logic from Single Cycle to MultiCycle



What Do We Want To Optimize

- **Single Cycle Architecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)
- **Single Cycle Architecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use the ALU for all operations (smaller size)
- **In Single Cycle Architecture all instructions take one cycle**
  - The most complex operation slows down everything!
  - Divide all instructions into multiple steps
  - Simpler instructions can take fewer cycles (average case may be faster)

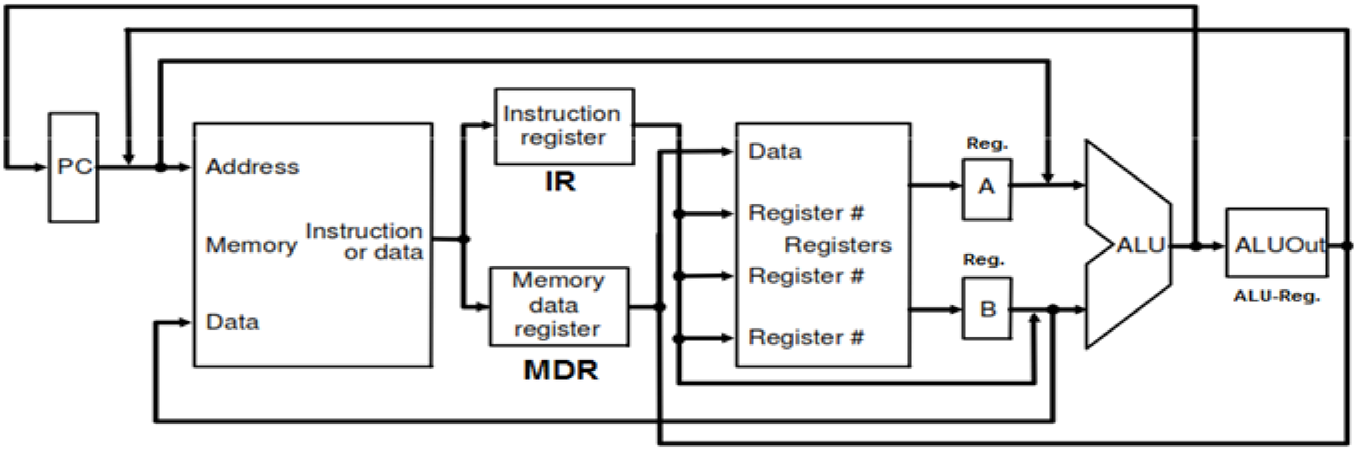


Multicycle Execution - Key Idea

- Break instruction execution into multiple cycles
- One clock cycle for each task
  1. Instruction Fetch
  2. Instruction Decode and Register Fetch
  3. Execution, memory address computation, or branch/jump completion
  4. Memory access / R-type instruction completion
  5. Memory read completion
- Share hardware to simplify datapath

Characteristics of Multicycle Design

- Instructions take more than one cycle
  - Some instructions take more cycles than others
  - Clock cycle is shorter than single-cycle clock
- Reuse of major components simplifies datapath
  - Single ALU for all calculations
  - Single memory for instructions and data
  - But, added registers needed to store values across cycles
- Control Unit Implemented by Finite State Machine
  - Control signals no longer a function of just the instruction.



- The multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help **reduce amount of hardware required**. The ability to allow instructions to take different numbers of clock cycles and the ability to share functional units within the execution of a single instruction are the major advantages of a multicycle design.

- The use of shared functional units requires the addition or widening of multiplexors as well as new temporary registers that hold data between clock cycles of the same instruction. **The additional registers are the: Instruction register (IR), Memory data Register (MDR) and A, B, and ALUOut.**

- The IR needs to hold the instruction until the end of execution of that instruction, and thus will require a **write control signal**. All the registers except the IR hold data only between a pair of adjacent clock cycles and will thus not need a write control signal.

❑ Between steps/cycles

✓ At the end of one cycle store data to be used in later cycles of the same instruction

- need to introduce additional internal (programmer-invisible) registers for this purpose

✓ Data to be used in later instructions are stored in programmer-visible state elements: the register file, PC, memory

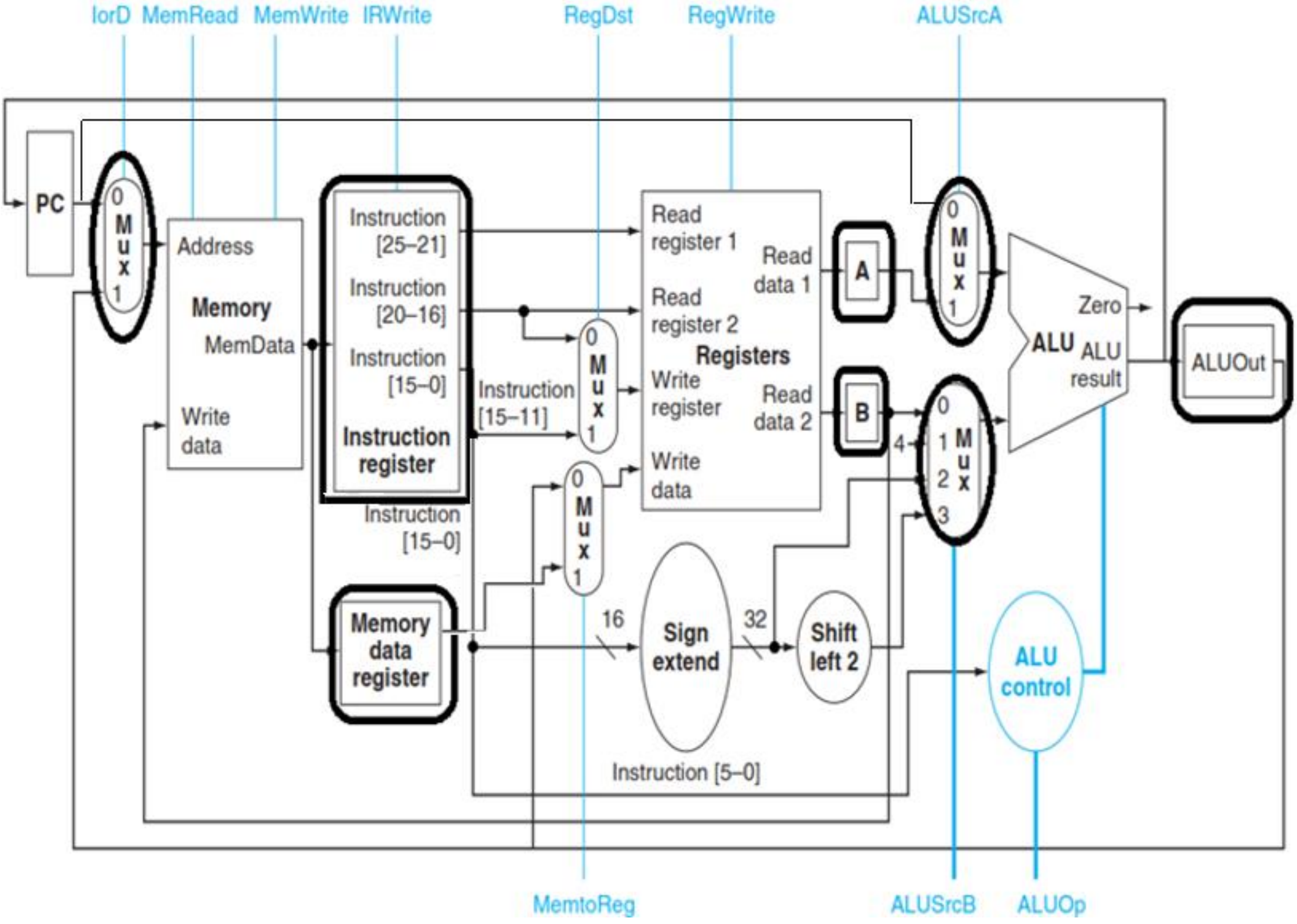
Multicycle Datapath for MIPS Handles the Basic Instructions

Handling the additional inputs requires two changes to the datapath:

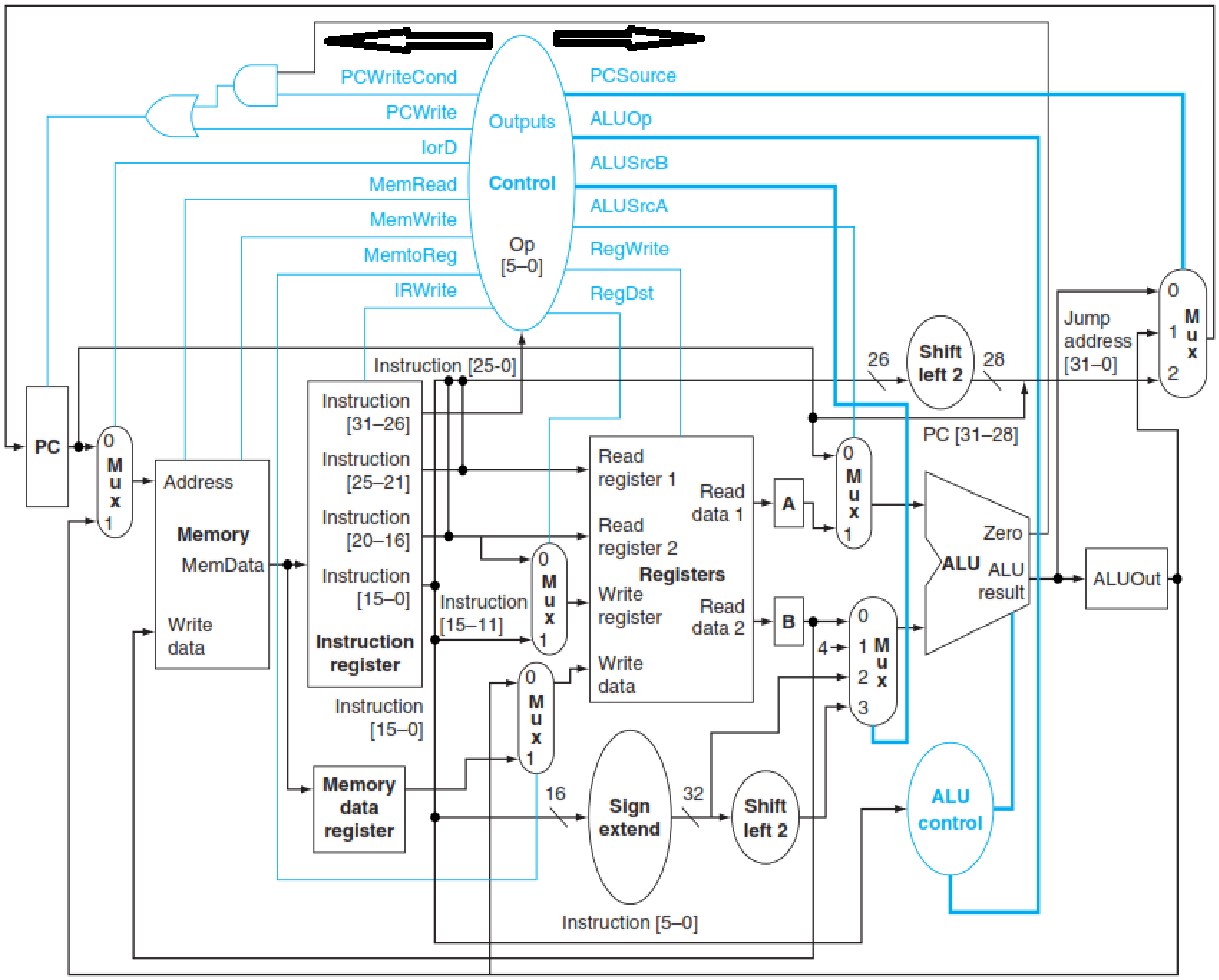
1. An additional multiplexor is added for the first ALU input. The multiplexor chooses between the A register and the PC.

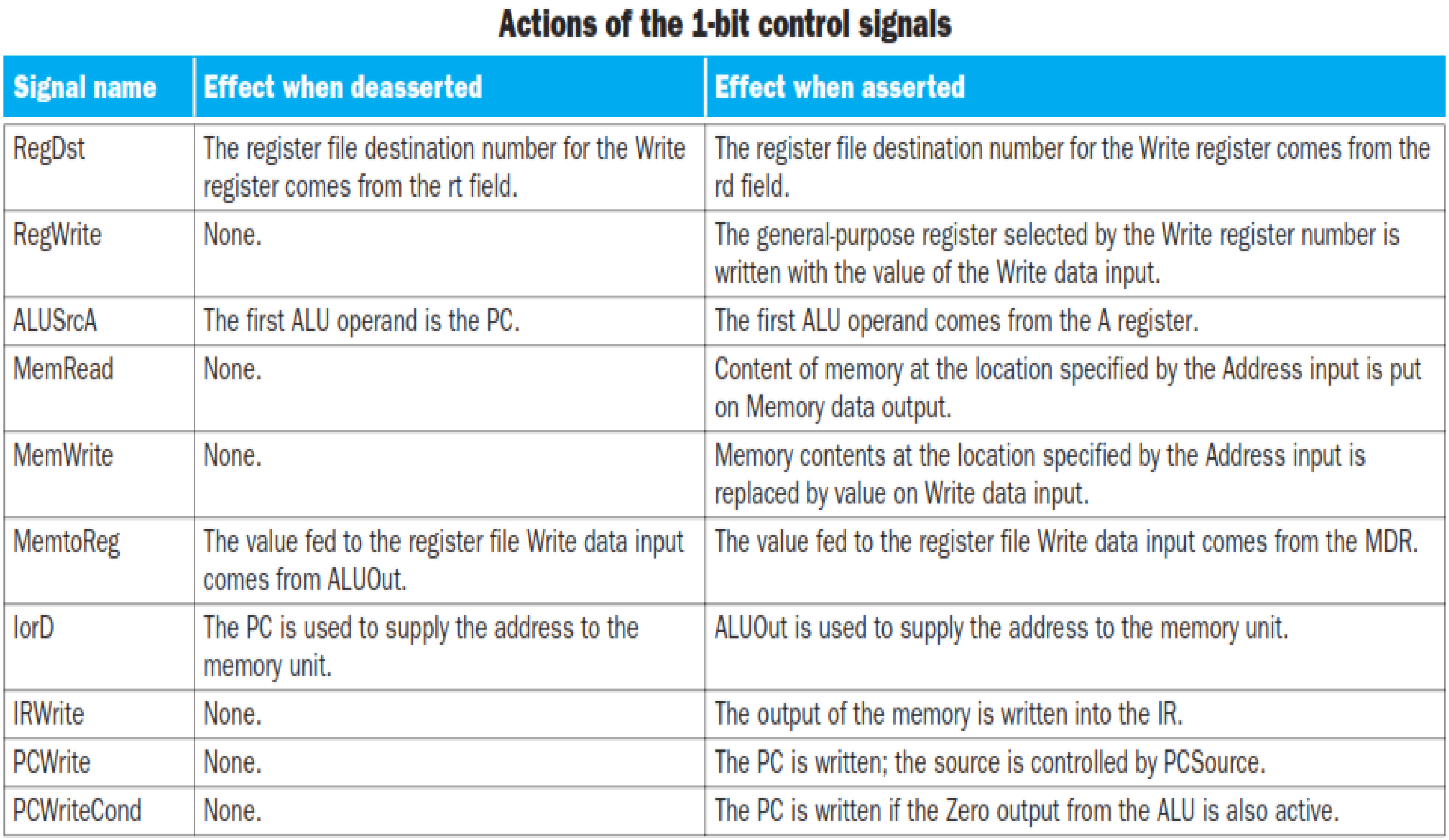
2. The multiplexor on the second ALU input is changed from a two–way to a four–way multiplexor. The two additional inputs to the multiplexor are the constant 4 (used to increment the PC) and the sign–extended and shifted offset field (used in the branch address computation).

3. By introducing a few registers and multiplexors, we are able to reduce the number of memory units from two to one and eliminate two adders. Since registers and multiplexors are fairly small compared to a memory unit or ALU, this could yield a substantial reduction in the hardware cost.



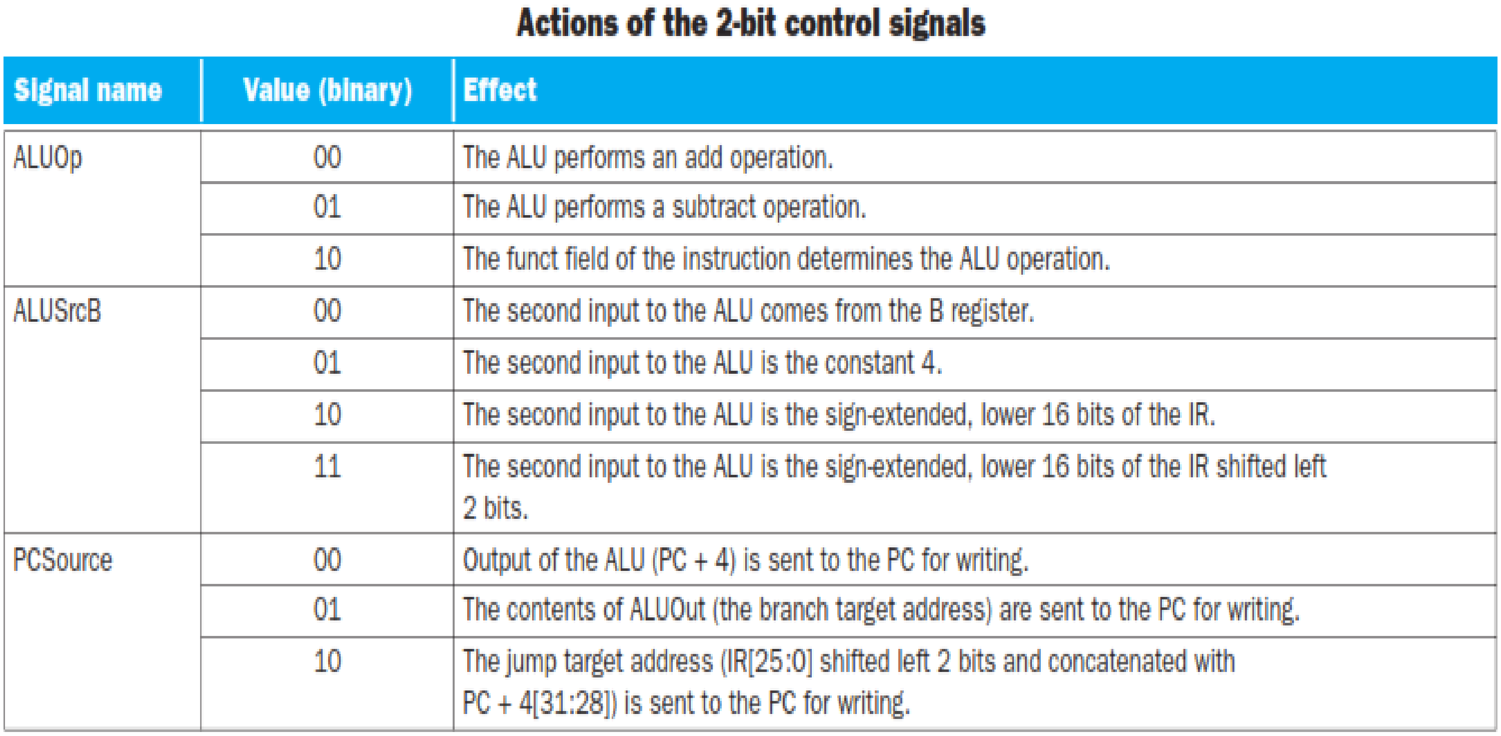
The complete datapath for the multicycle implementation together with the necessary control lines.





Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

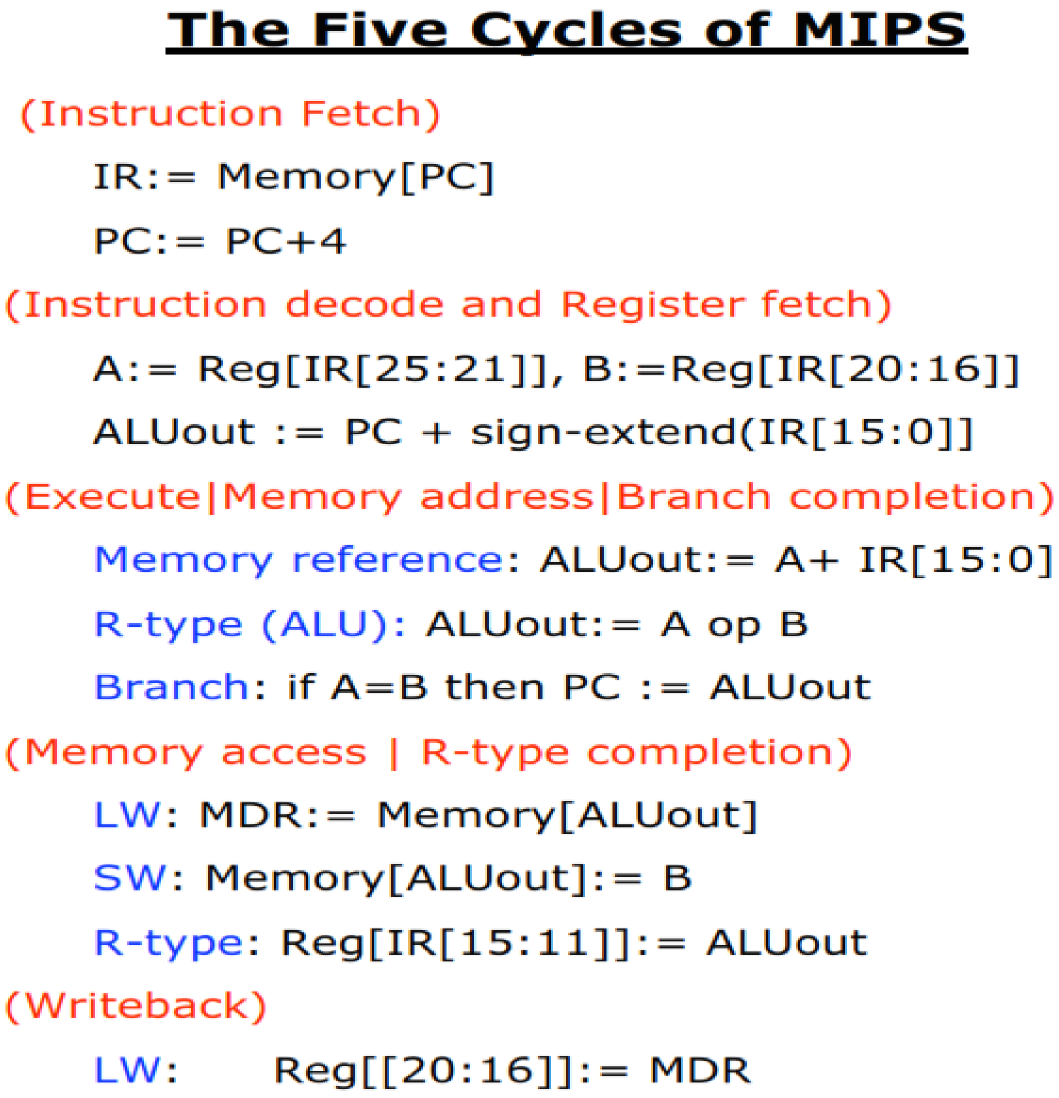


Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

Five Stages of Instruction Execution

1. Instruction Fetch and PC increment
2. Instruction Decode and Register Fetch (and branch target calculation)
3. One of the following:
   - Execute R-Type Instruction OR Calculate memory address for load/store OR Perform comparison for branch OR Jump completion.
4. Memory access for load/store OR R-type instruction completion (save result)
5. Memory read completion (save result - load only)



The Five Cycles of MIPS

(Instruction Fetch)

IR:= Memory[PC]

PC:= PC+4

(Instruction decode and Register fetch)

A:= Reg[IR[25:21]], B:=Reg[IR[20:16]]

ALUout := PC + sign-extend(IR[15:0]]

(Execute|Memory address|Branch completion)

Memory reference: ALUout:= A+ IR[15:0]

R-type (ALU): ALUout:= A op B

Branch: if A=B then PC := ALUout

(Memory access | R-type completion)

LW: MDR:= Memory[ALUout]

SW: Memory[ALUout]:= B

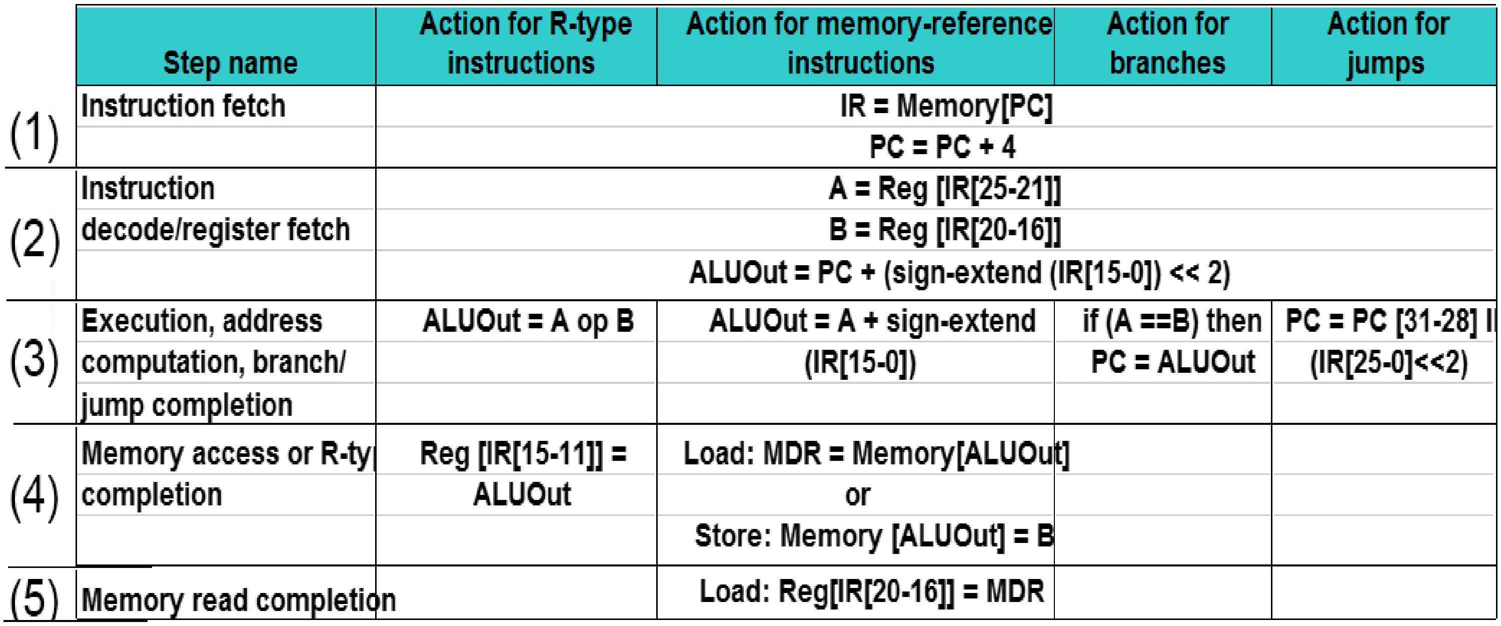
R-type: Reg[IR[15:11]]:= ALUout

(Writeback)

LW: Reg[[20:16]]:= MDR

Notes:-

- Not all instructions require all the steps.
- Each step takes one clock cycle.
- Each MIPS instruction takes from 3 – 5 cycles (steps).



	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
(1)	Instruction fetch	IR = Memory[PC] PC = PC + 4			
(2)	Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
(3)	Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] I (IR[25-0]<<2)
(4)	Memory access or R-ty completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
(5)	Memory read completion		Load: Reg[IR[20-16]] = MDR		

Why intermediate registers?

Sometimes we need the output of a functional unit in a later clock cycle during the execution of an instruction.

(Example: The instruction word fetched in stage 1 determines the destination of the register write in stage 5. The ALU result for an address computation in stage 3 is needed as the memory address for lw or sw in stage 4.)

These outputs must be stored in intermediate registers for future use. Otherwise they will be lost by the next clock cycle.

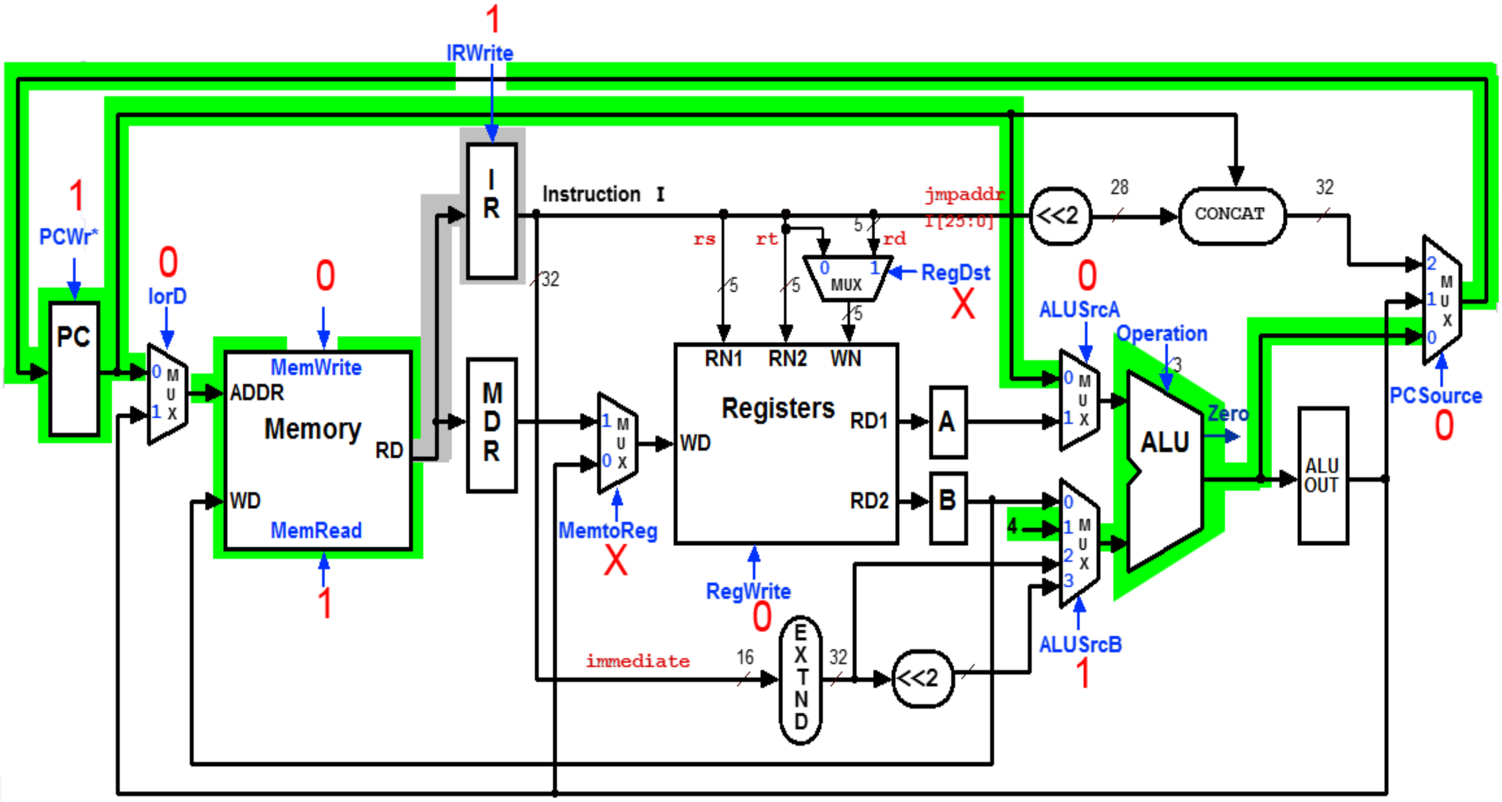
(Instruction read in stage 1 is saved in Instruction register. Register file outputs from stage 2 are saved in registers A and B. The ALU output will be stored in a register ALUout. Any data fetched from memory in stage 4 is kept in the Memory data register MDR.)

STEP 1

- ❑ Instruction Fetch & PC Increment (**IF**):
- ❑ Use PC to get instruction and put it in the instruction register.
- ❑ Increment the PC by 4 and put the result back in the PC.
- ❑ Can be described using *RTL* (*Register-Transfer Language*):

IR = Memory[PC];

PC = PC + 4;



STEP 2

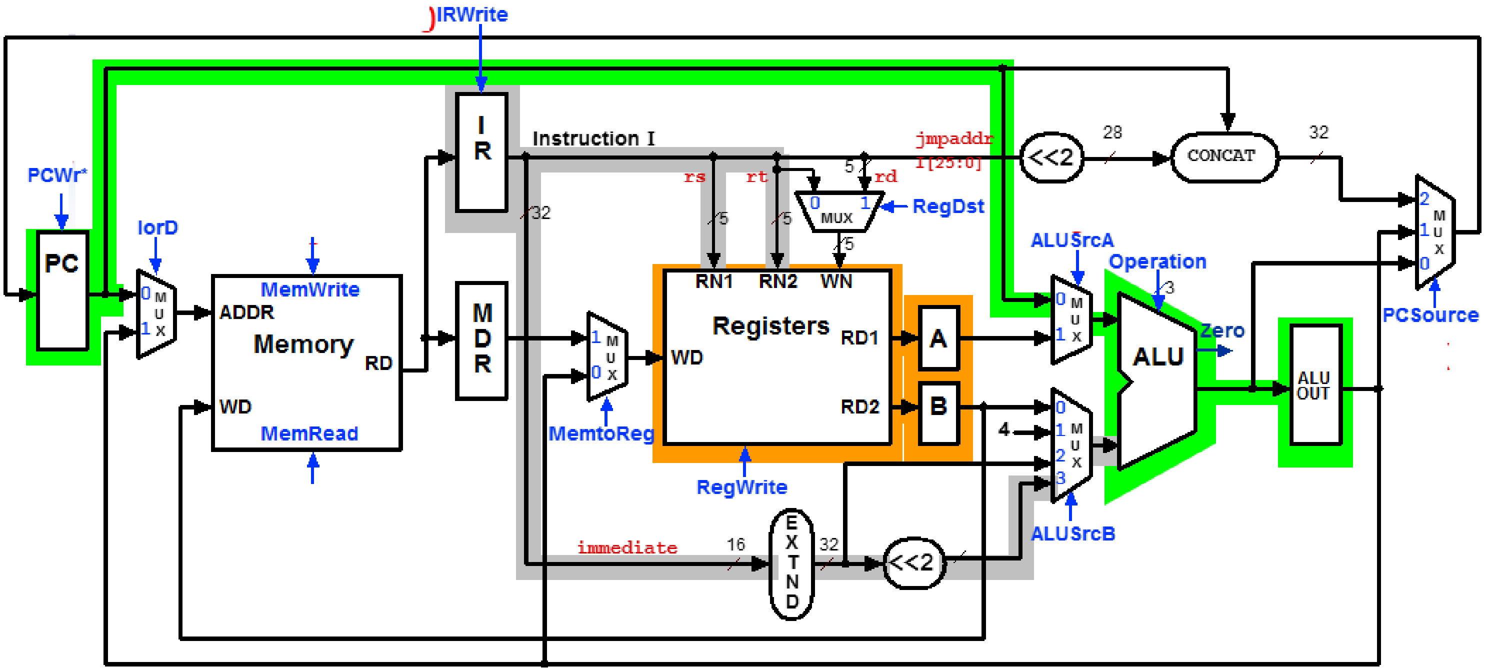
❑ Instruction Decode and Register Fetch (**ID**):

❑ Read registers rs and rt in case we need them.

❑ Compute the branch address in case the instruction is a branch.

❑ RTL:

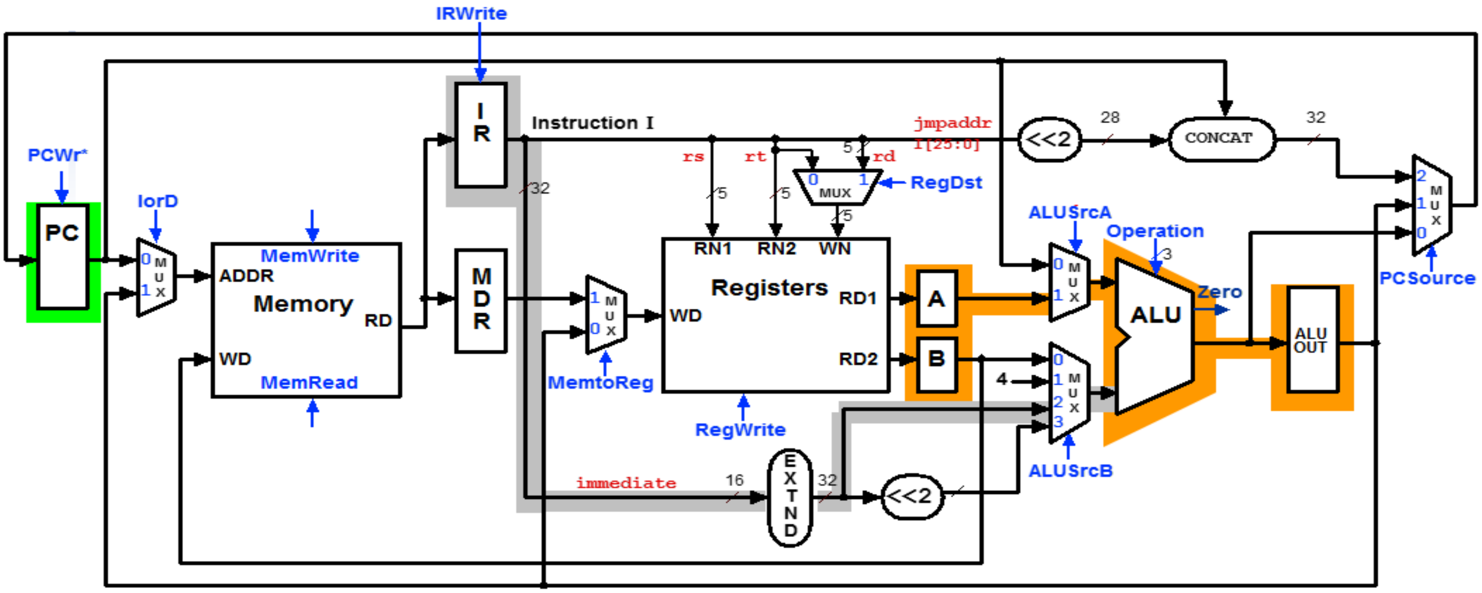
```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```



STEP 3

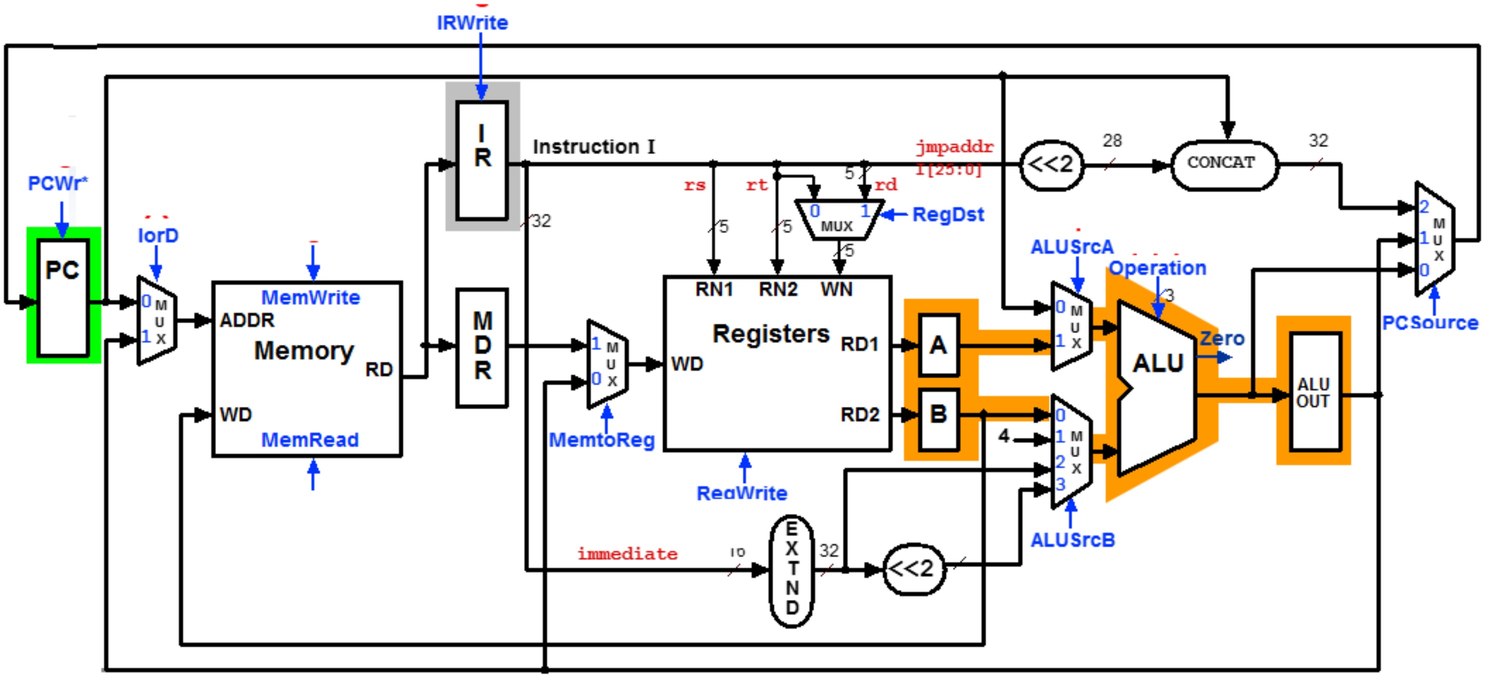
- ❑ Execution, Address Computation or Branch Completion(**EX**):
- ❑ ALU performs one of four functions *depending* on instruction type:

1. memory reference: `ALUOut = A + sign-extend(IR[15-0]);`



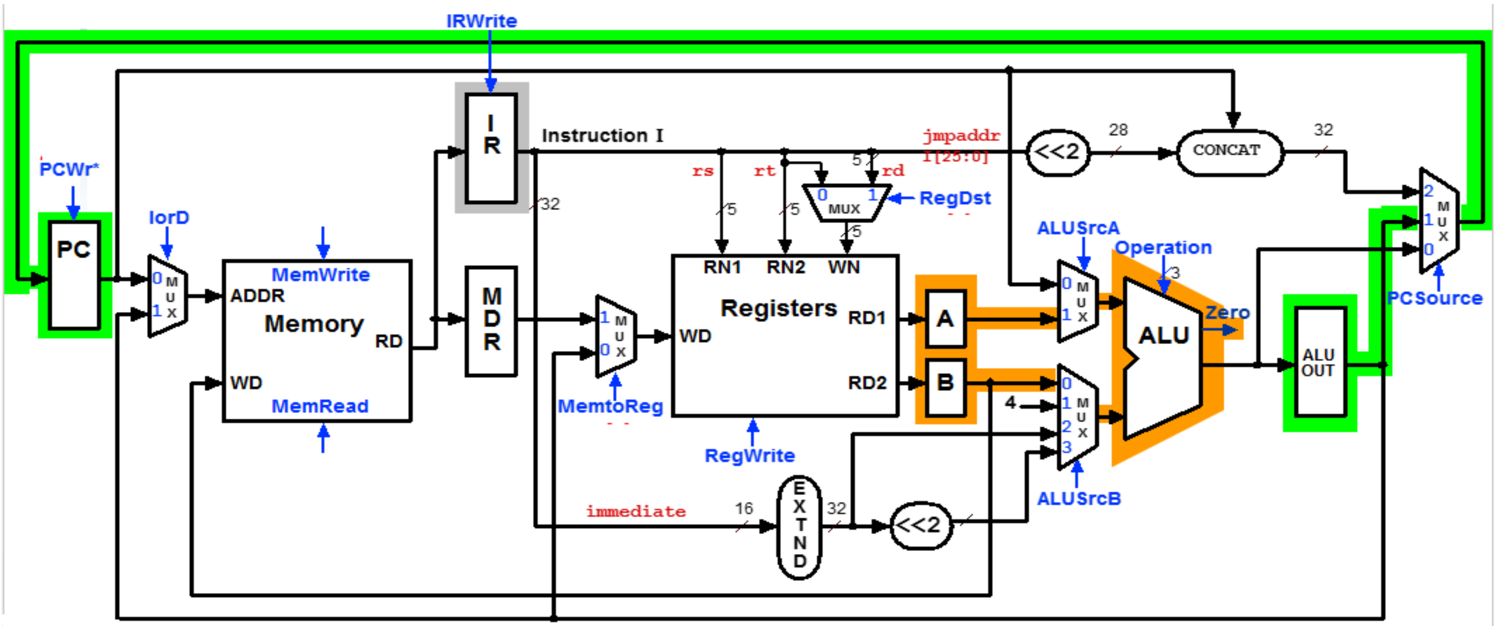
- ❑ Execution, Address Computation or Branch Completion(**EX**):

2. R-type: `ALUOut = A op B;`



❑ Execution, Address Computation or Branch Completion(**EX**):

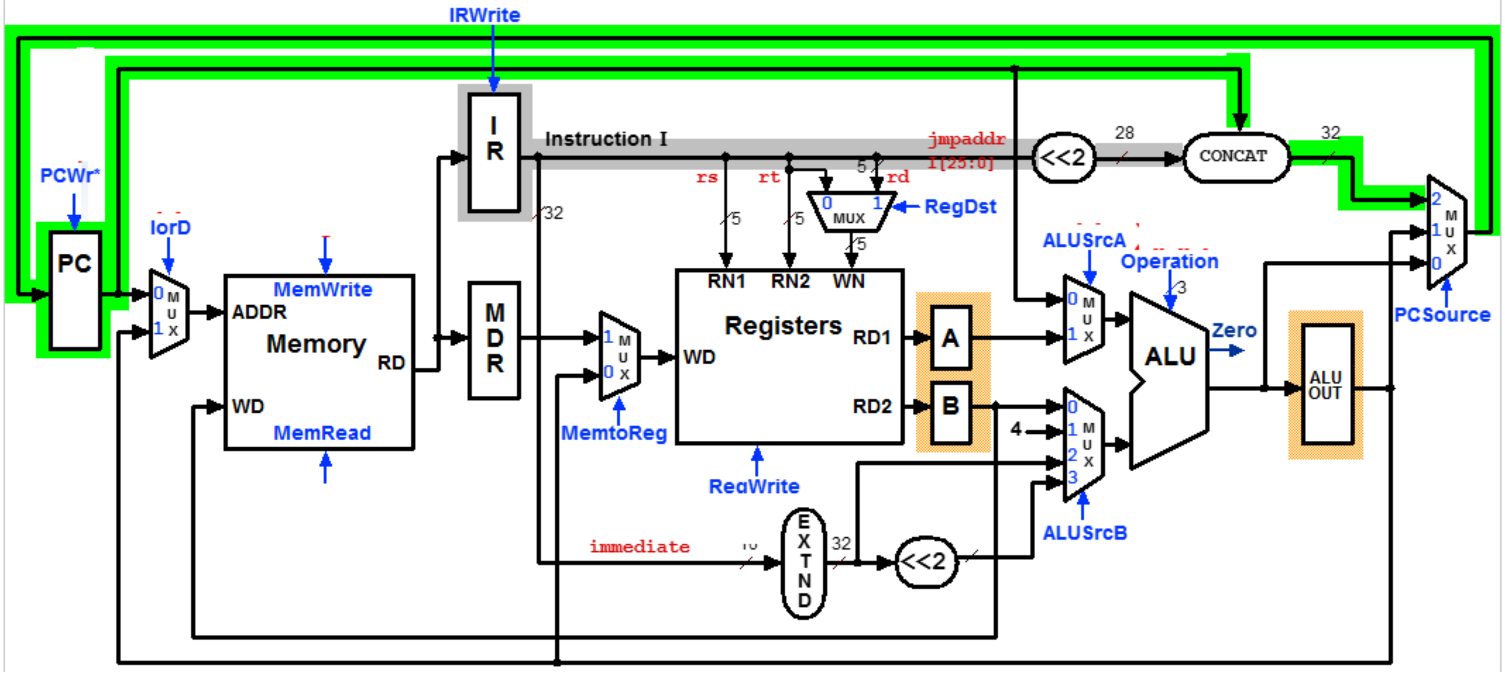
3. branch (*instruction completes*): `if (A==B) PC = ALUOut;`



❑ Execution, Address Computation or Branch Completion(**EX**):

4. jump (*instruction completes*): `PC = PC[31-28] || (IR(25-0) << 2)`

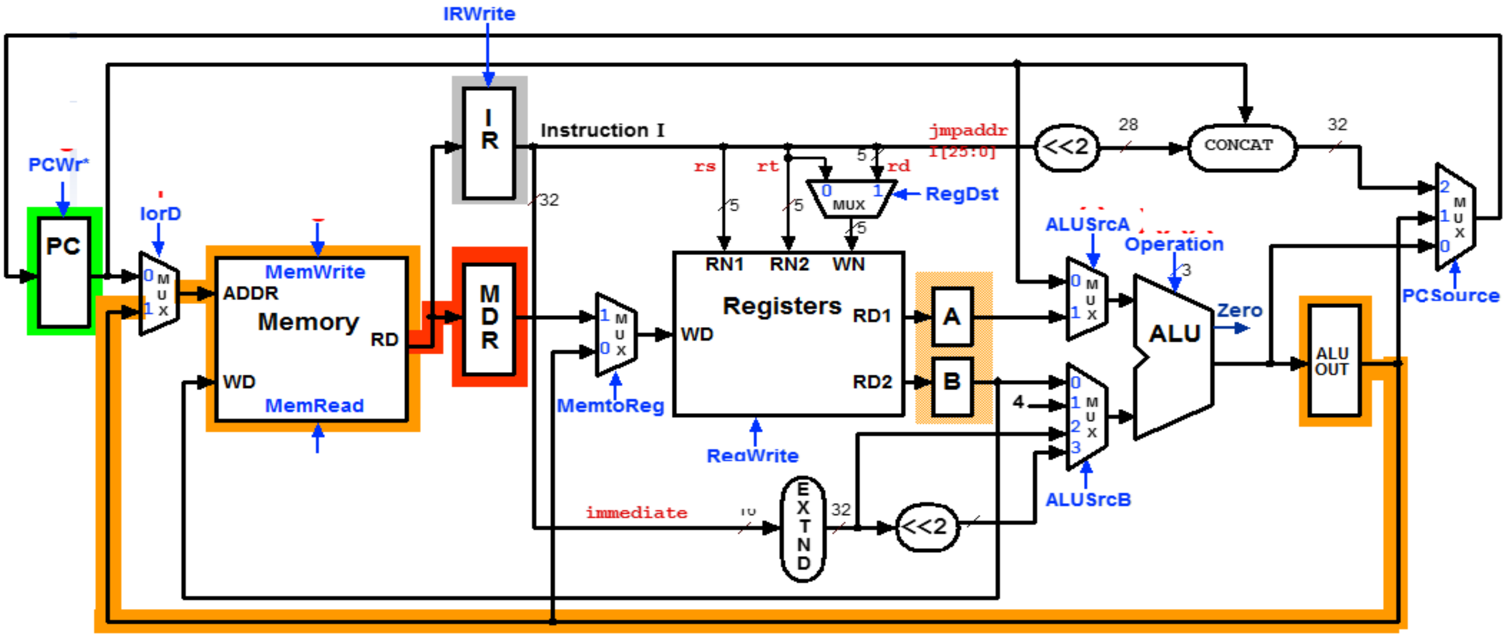
PC = PC[31-28] concat (IR[25-0] << 2)



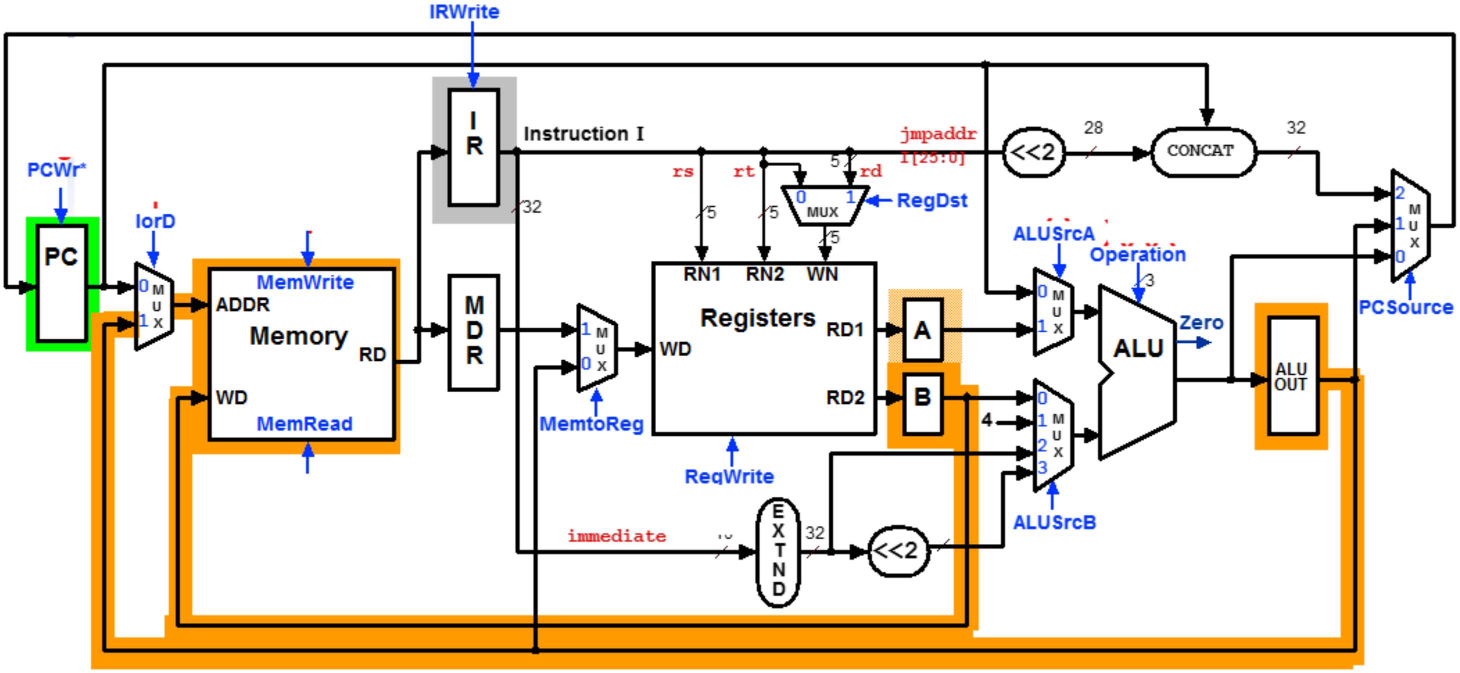
STEP 4

- ❑ Memory access or R-type Instruction Completion (**MEM**):
- ❑ Again *depending* on instruction type:

1. Loads and stores access memory
    - ❑ Load: `MDR = Memory[ALUOut];`



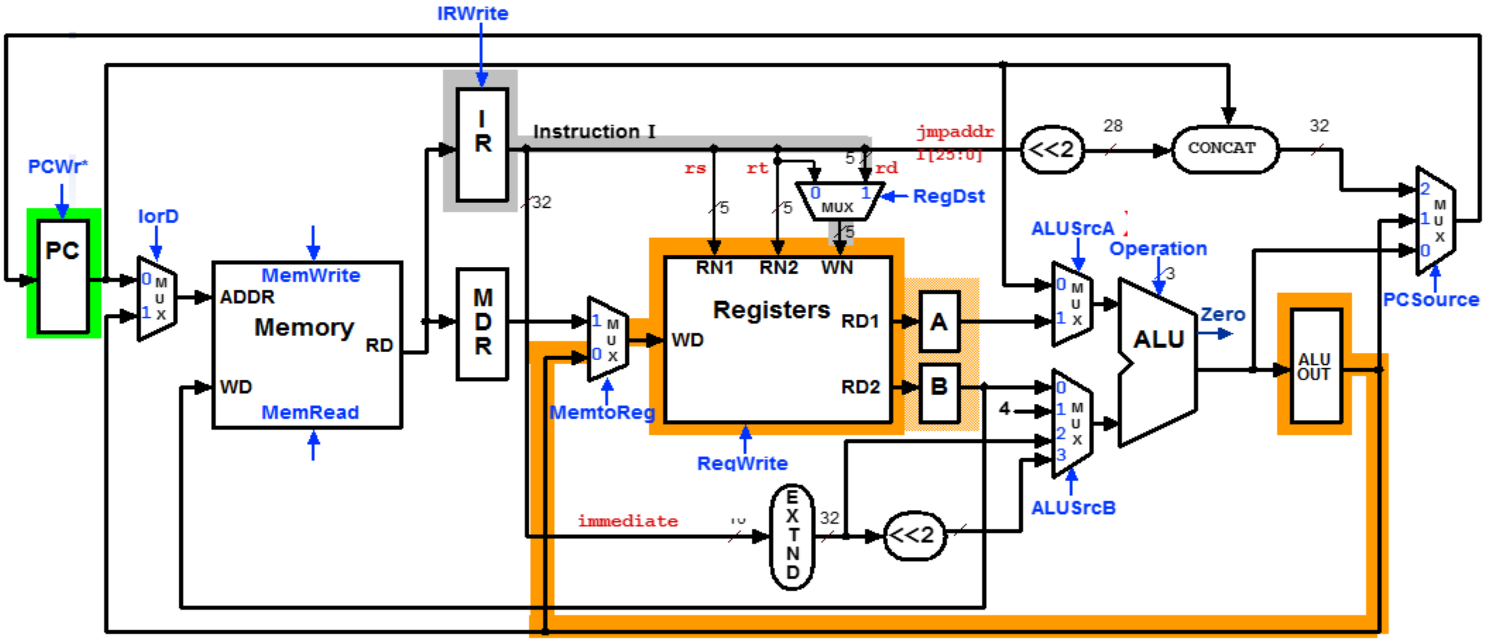
❑ Store (*instruction completes*): `Memory[ALUOut] = B;`



❑ Memory access or R-type Instruction Completion (**MEM**):

2. R-type (*instruction completes*): `Reg[IR[15-11]] = ALUOut;`

`Reg[IR[15:11]] = ALUOut;` `(Reg[Rd] = ALUOut)`

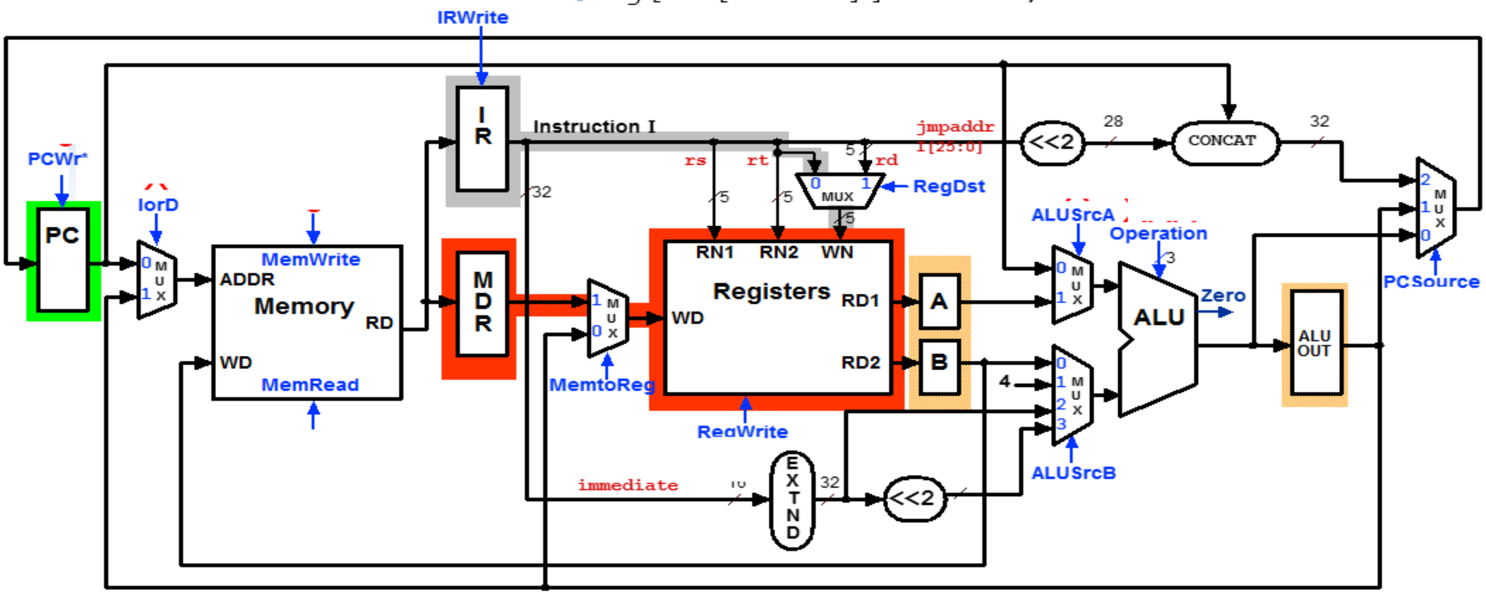


STEP 5

❑ Memory Read Completion (**WB**):

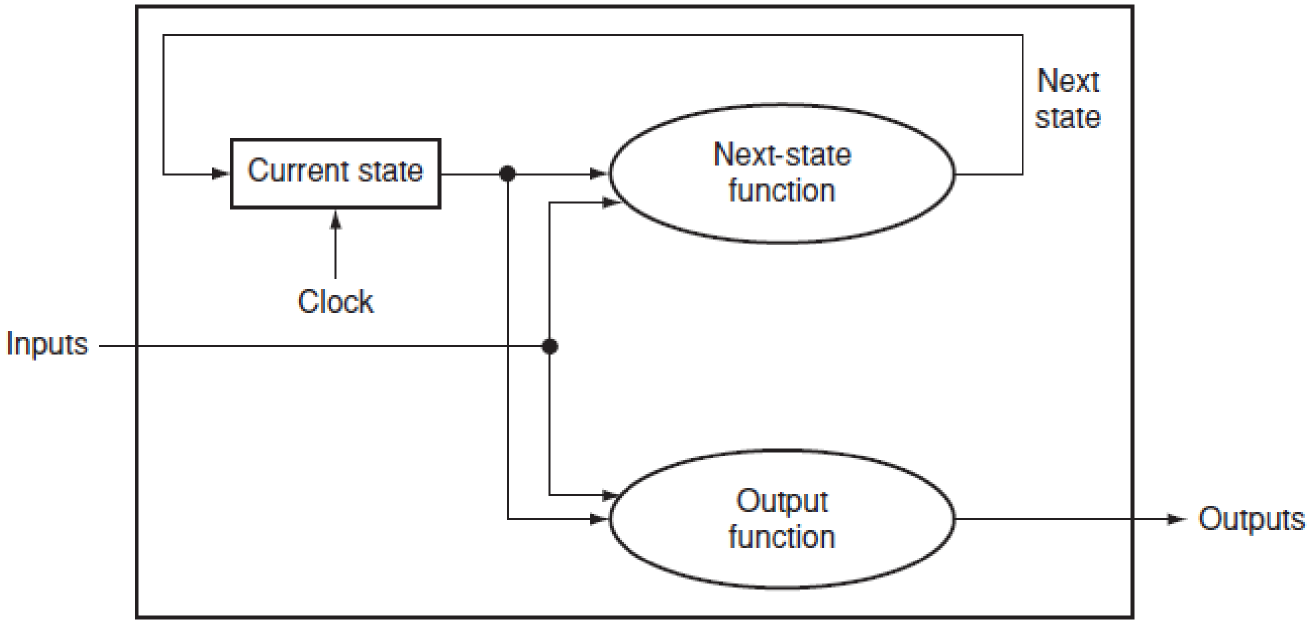
❑ Load writes back (instruction *completes*) `Reg[IR[20-16]]= MDR;`

`Reg[IR[20-16]] = MDR;`



Review: Finite-State Machines

- Digital logic systems can be classified as **combinational** or **sequential**.
- Sequential systems contain state stored in memory elements internal to the system. Their behavior depends both on the set of inputs supplied and on the contents of the internal memory, or state of the system.
- Thus, a sequential system cannot be described with a truth table. Instead, a sequential system is described as a **finite-state machine** (or often just **state machine**).
- A **finite-state machine** has a set of states and two functions called the **next-state function** and the **output function.**
- **finite-state machine** : A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.
- **next-state function** : A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.

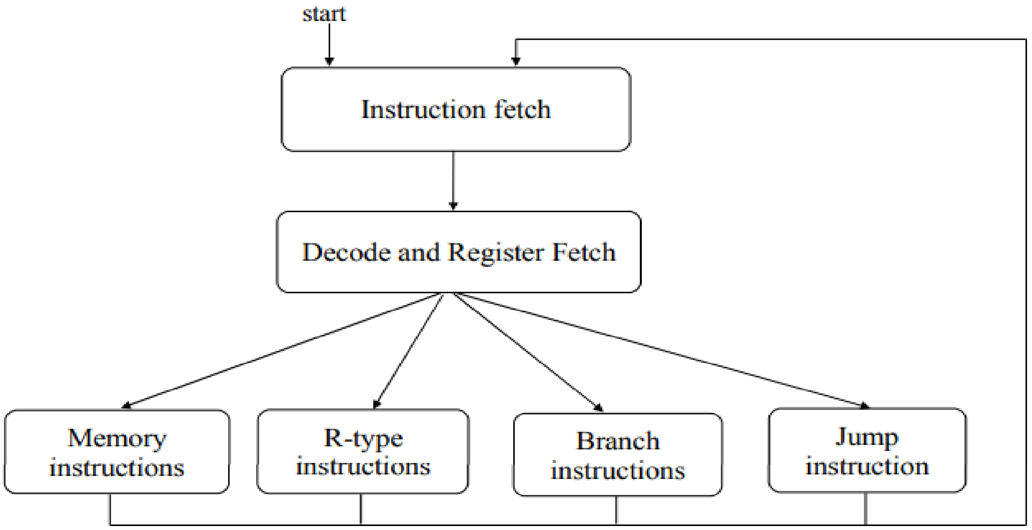


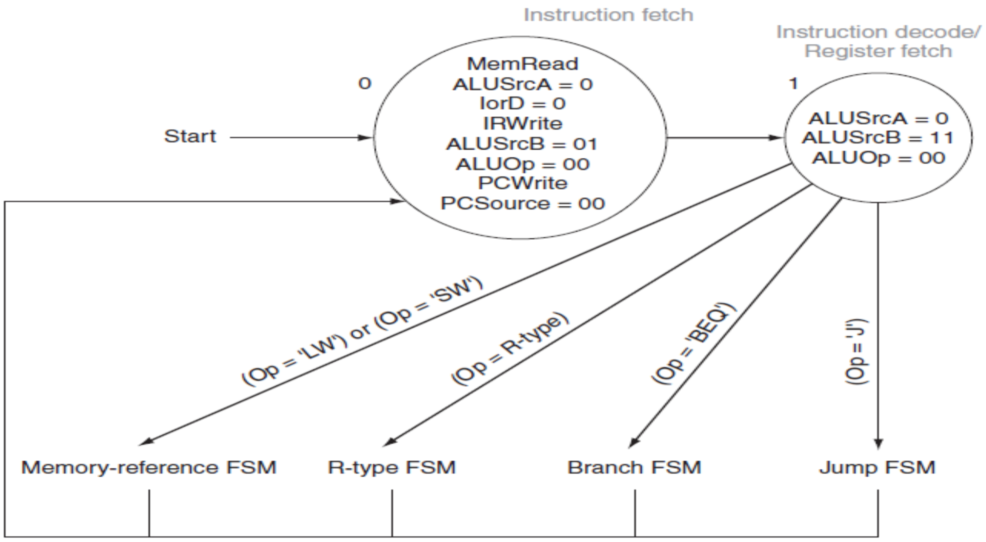
A state machine consists of internal storage that contains the state and two combinational functions: the next-state function and the output function. Often, the output function is restricted to take only the current state as its input; this does not change the capability of a sequential machine, but does affect its internals.

Multicycle Control

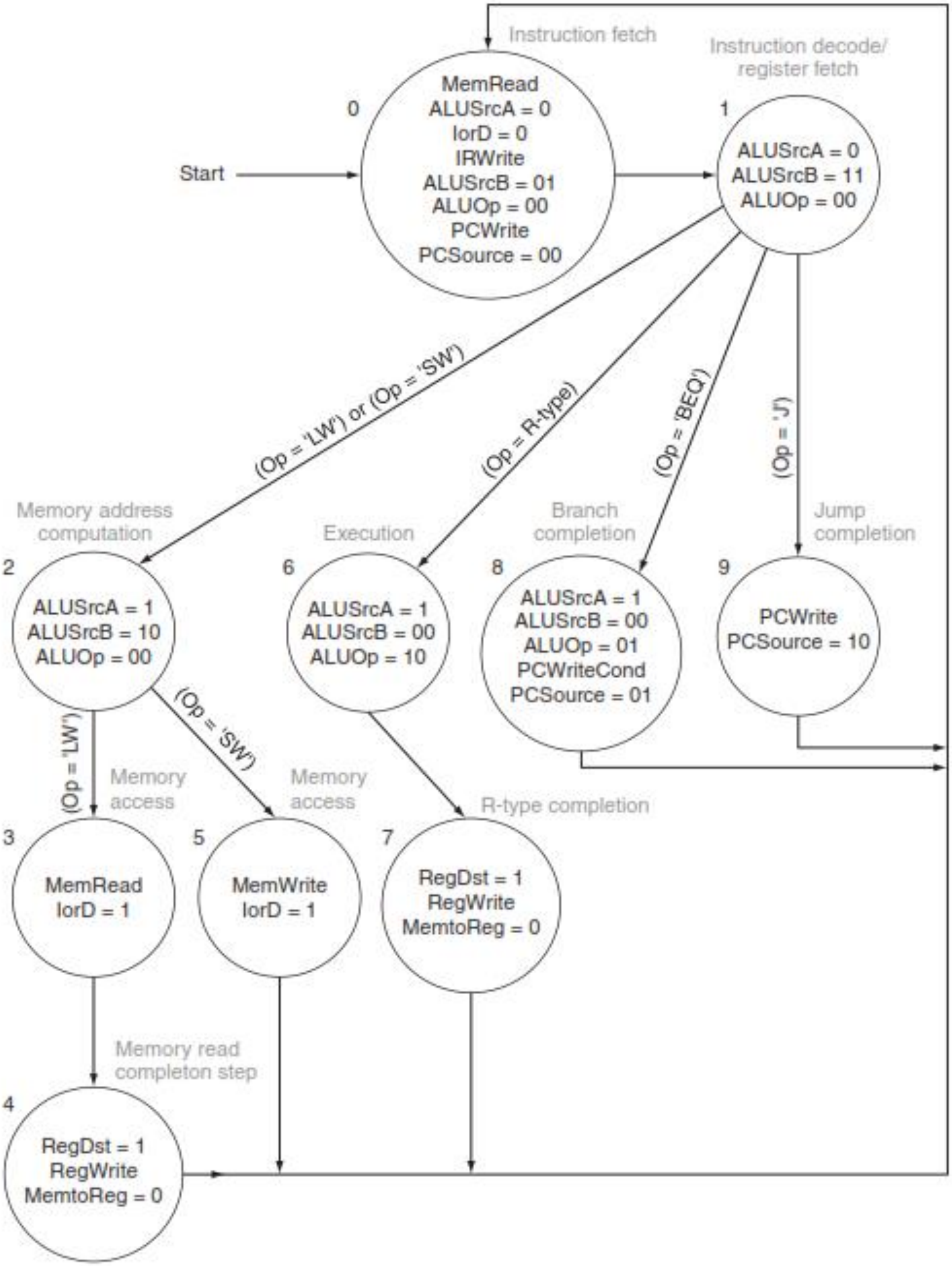
- Single-cycle control used combinational logic
- Multi-cycle control uses ??
- FSM defines a succession of states, transitions between states (based on inputs), and outputs (based on state)
- First two states same for every instruction, next state depends on opcode

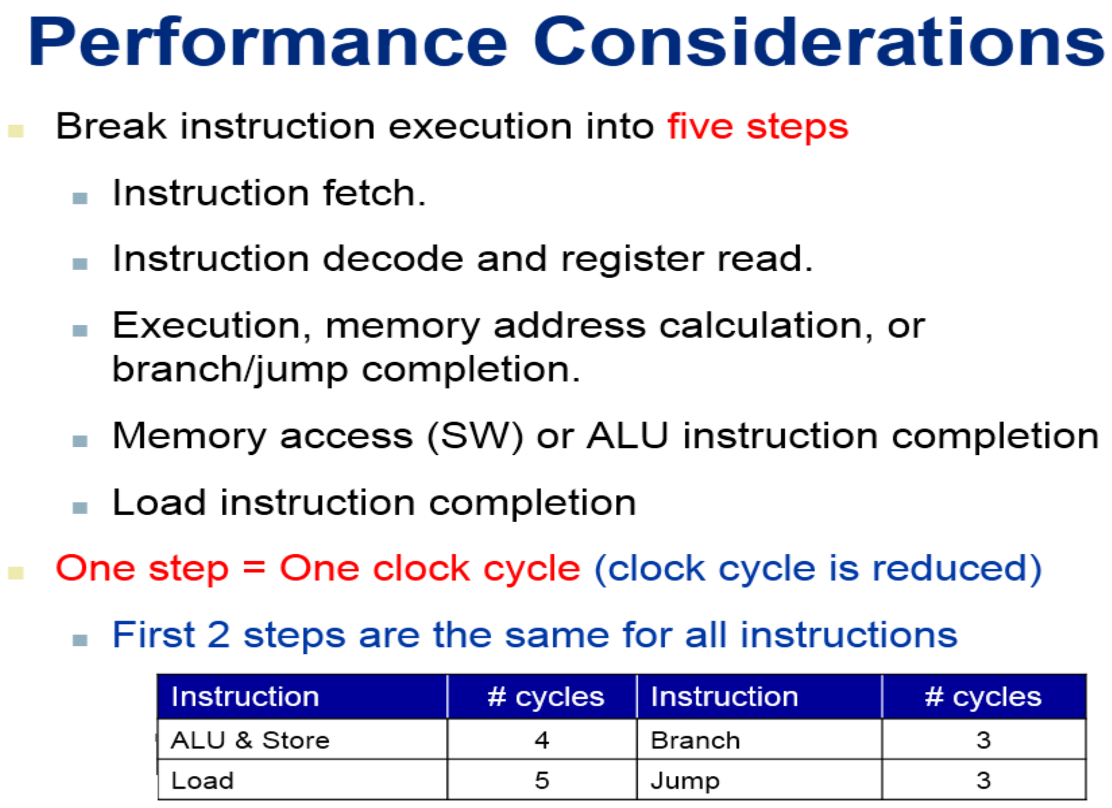
Multicycle Control FSM





Complete finite State Machine Control for The Datapath

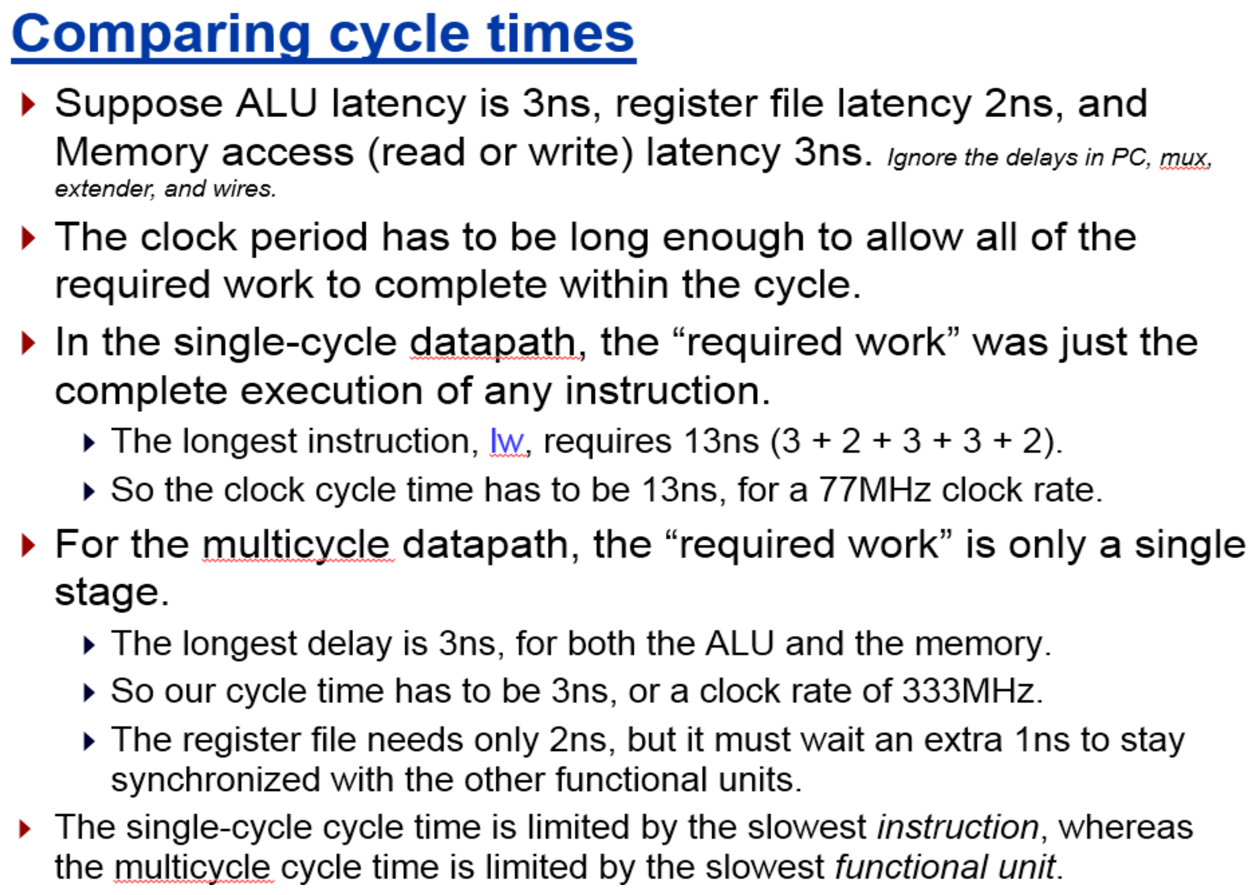

Instruction fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
Instruction decode/
register fetch
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
Memory
access
(Op = 'SW')
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Memory read
completon step
4
RegDst = 1
RegWrite
MemtoReg = 0



Performance Considerations

- Break instruction execution into five steps
  - Instruction fetch.
  - Instruction decode and register read.
  - Execution, memory address calculation, or branch/jump completion.
  - Memory access (SW) or ALU instruction completion
  - Load instruction completion
- One step = One clock cycle (clock cycle is reduced)
  - First 2 steps are the same for all instructions

Instruction	# cycles	Instruction	# cycles
ALU & Store	4	Branch	3
Load	5	Jump	3

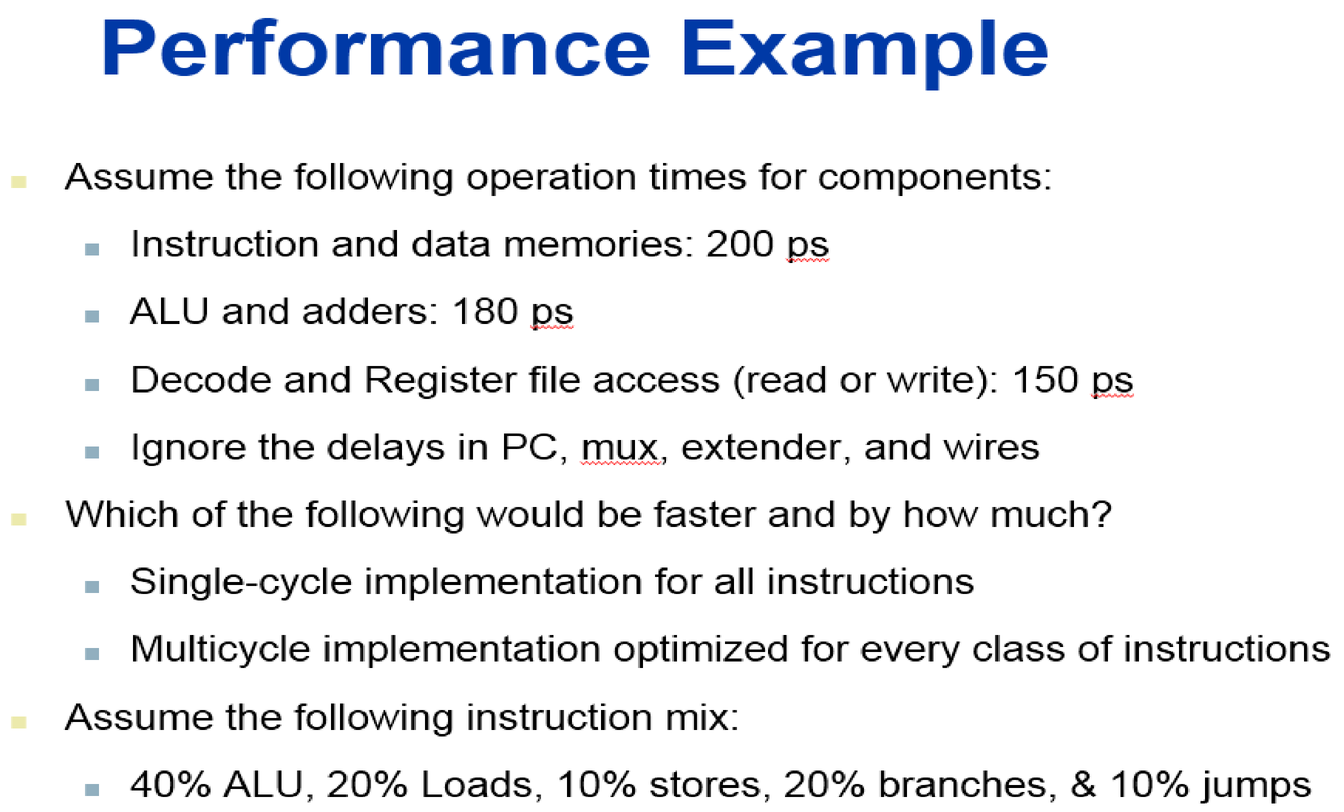


Comparing cycle times

- Suppose ALU latency is 3ns, register file latency 2ns, and Memory access (read or write) latency 3ns. *Ignore the delays in PC, mux, extender, and wires.*
- The clock period has to be long enough to allow all of the required work to complete within the cycle.
- In the single-cycle datapath, the “required work” was just the complete execution of any instruction.
  - The longest instruction, lw, requires 13ns (3 + 2 + 3 + 3 + 2).
  - So the clock cycle time has to be 13ns, for a 77MHz clock rate.
- For the multicycle datapath, the “required work” is only a single stage.
  - The longest delay is 3ns, for both the ALU and the memory.
  - So our cycle time has to be 3ns, or a clock rate of 333MHz.
  - The register file needs only 2ns, but it must wait an extra 1ns to stay synchronized with the other functional units.
- The single-cycle cycle time is limited by the slowest *instruction*, whereas the multicycle cycle time is limited by the slowest *functional unit*.

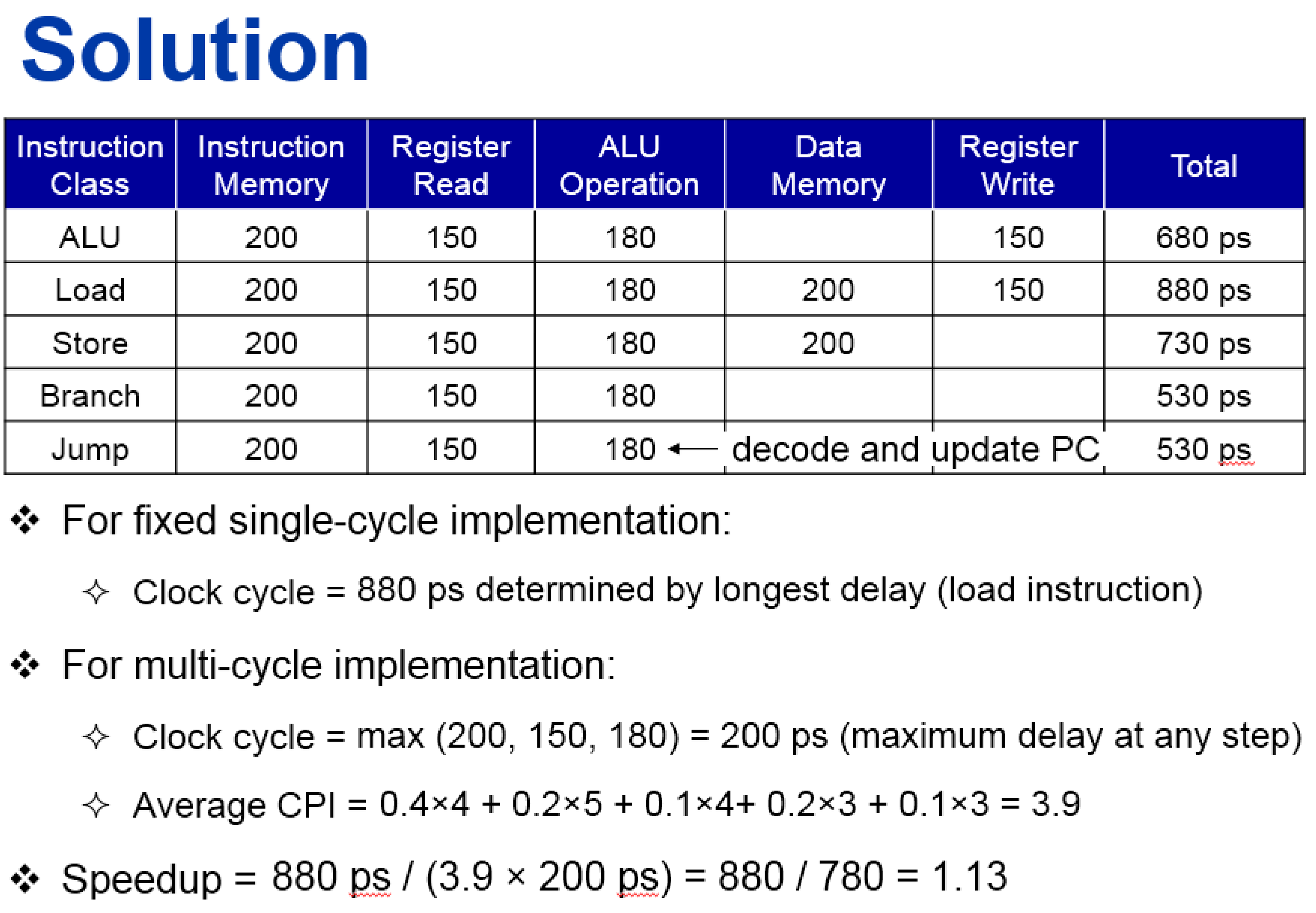
Comparing instruction execution times

- In the single-cycle datapath, each instruction needs an entire clock cycle, or 13ns, to execute.
- With the multicycle CPU, different instructions need different numbers of clock cycles, and hence different amounts of time.
  - A branch needs 3 cycles, or 3 x 3ns = 9ns.
  - Arithmetic and sw instructions each require 4 cycles, or 12ns.
  - Finally, a lw takes 5 stages, or 15ns.
- We can make some observations about performance already.
  - Loads take *longer* with this multicycle implementation, while all other instructions are faster than before.
  - So if our program doesn't have too many loads, then we should see an increase in performance.



Performance Example

- Assume the following operation times for components:
  - Instruction and data memories: 200 ps
  - ALU and adders: 180 ps
  - Decode and Register file access (read or write): 150 ps
  - Ignore the delays in PC, mux, extender, and wires
- Which of the following would be faster and by how much?
  - Single-cycle implementation for all instructions
  - Multicycle implementation optimized for every class of instructions
- Assume the following instruction mix:
  - 40% ALU, 20% Loads, 10% stores, 20% branches, & 10% jumps



Solution

Instruction Class	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	Total
ALU	200	150	180		150	680 ps
Load	200	150	180	200	150	880 ps
Store	200	150	180	200		730 ps
Branch	200	150	180			530 ps
Jump	200	150	180 ←	decode and update PC		530 ps

❖ For fixed single-cycle implementation:

✧ Clock cycle = 880 ps determined by longest delay (load instruction)

❖ For multi-cycle implementation:

✧ Clock cycle = max (200, 150, 180) = 200 ps (maximum delay at any step)

✧ Average CPI = 0.4×4 + 0.2×5 + 0.1×4+ 0.2×3 + 0.1×3 = 3.9

❖ Speedup = 880 ps / (3.9 × 200 ps) = 880 / 780 = 1.13

example

- Let’s assume the instruction mix.

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

- In a single-cycle datapath, all instructions take 13ns to execute.
- The average execution time for an instruction on the multicycle processor works out to 12.09ns.

  (48% x 12ns) + (22% x 15ns) + (11% x 12ns) + (19% x 9ns) = 12.09ns

- The multicycle implementation is faster in this case, but not by much. The speedup here is only 13ns / 12.09ns = 1.075

Overview of a Multiple Cycle Implementation

° **The root of the single cycle processor’s problems:**

- The cycle time has to be long enough for the slowest instruction

° **Solution:**

- Break the instruction into smaller steps.
- Execute each step (instead of the entire instruction) in one cycle

- **Cycle time**: time it takes to execute the longest step

- Keep all the steps to have similar length

- This is the essence of the multiple cycle processor

° **The advantages of the multiple cycle processor:**

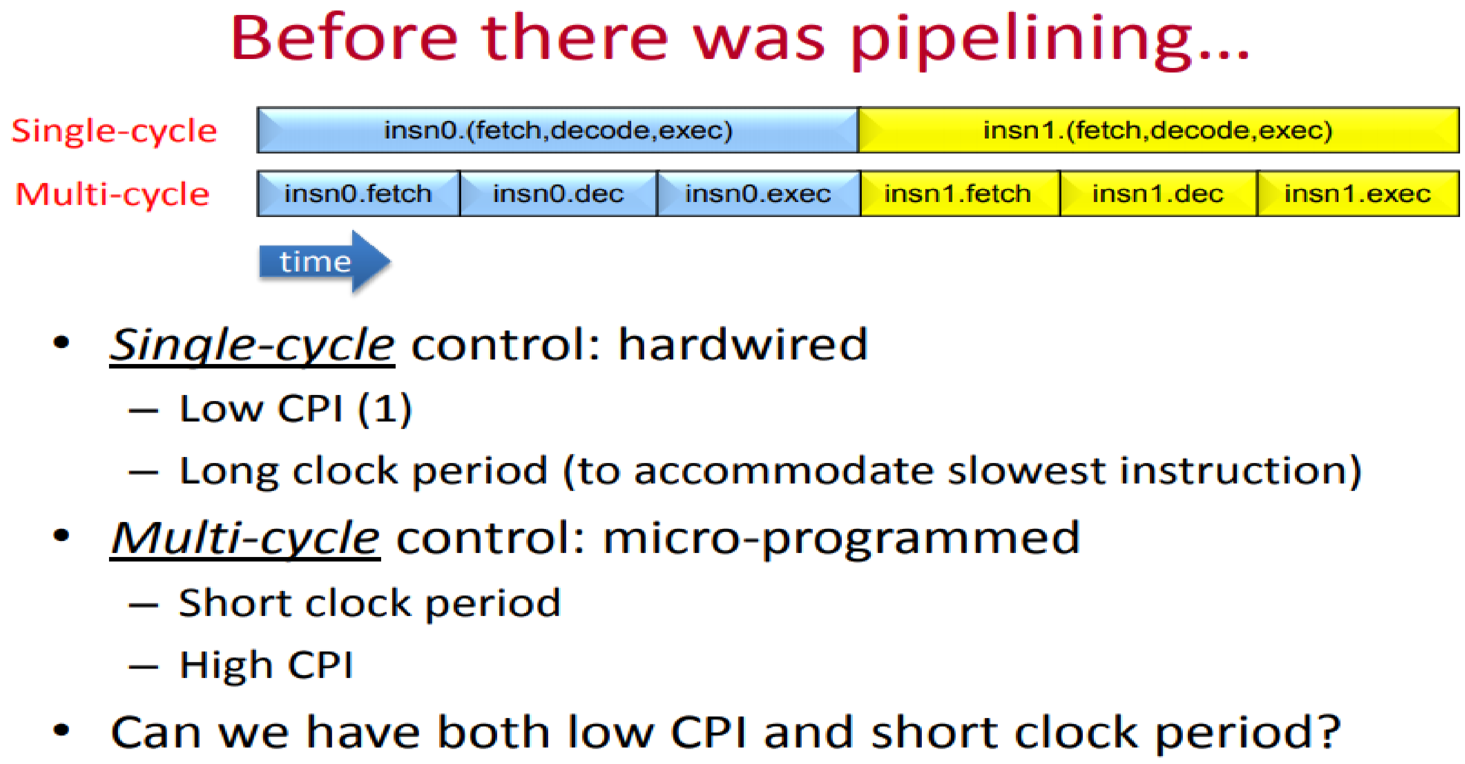
- Cycle time is much shorter
- Different instructions take different number of cycles to complete - Load takes five cycles - Jump only takes three cycles
- Allows a functional unit to be used more than once per instruction

Pipelining Processor Design

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure.

General Principles of Pipelining

- Express task as a collection of stages
- Move instructions through stages
- Process several instructions at any given moment



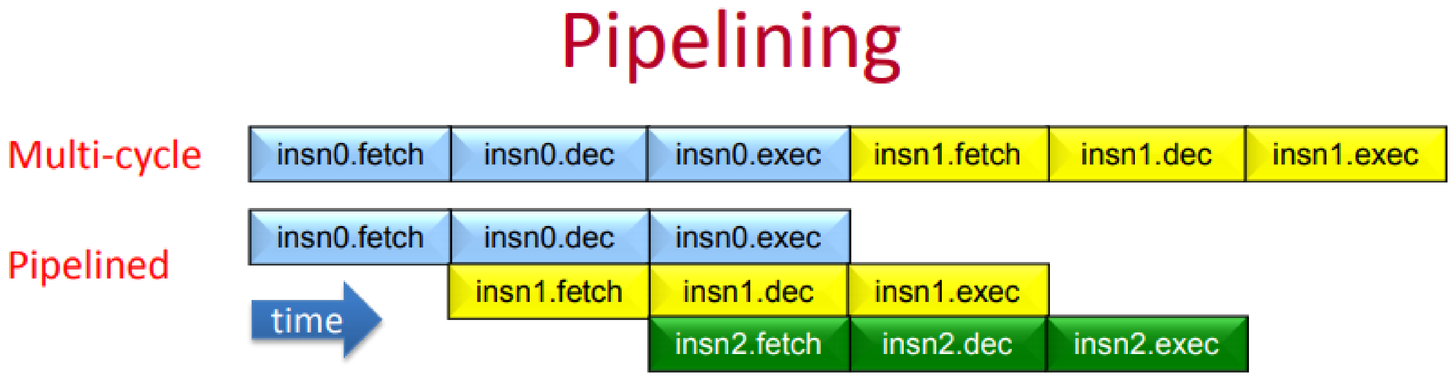
Before there was pipelining…

Single-cycle: insn0.(fetch,decode,exec) | insn1.(fetch,decode,exec)

Multi-cycle: insn0.fetch | insn0.dec | insn0.exec | insn1.fetch | insn1.dec | insn1.exec

time

- *Single-cycle* control: hardwired
  - Low CPI (1)
  - Long clock period (to accommodate slowest instruction)
- *Multi-cycle* control: micro-programmed
  - Short clock period
  - High CPI
- Can we have both low CPI and short clock period?



Pipelining

Multi-cycle: insn0.fetch | insn0.dec | insn0.exec | insn1.fetch | insn1.dec | insn1.exec

Pipelined:
insn0.fetch | insn0.dec | insn0.exec
insn1.fetch | insn1.dec | insn1.exec
insn2.fetch | insn2.dec | insn2.exec

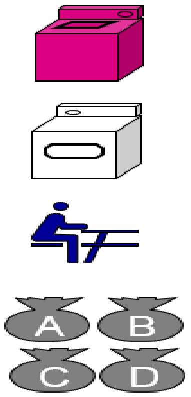
time

Pipelining Example

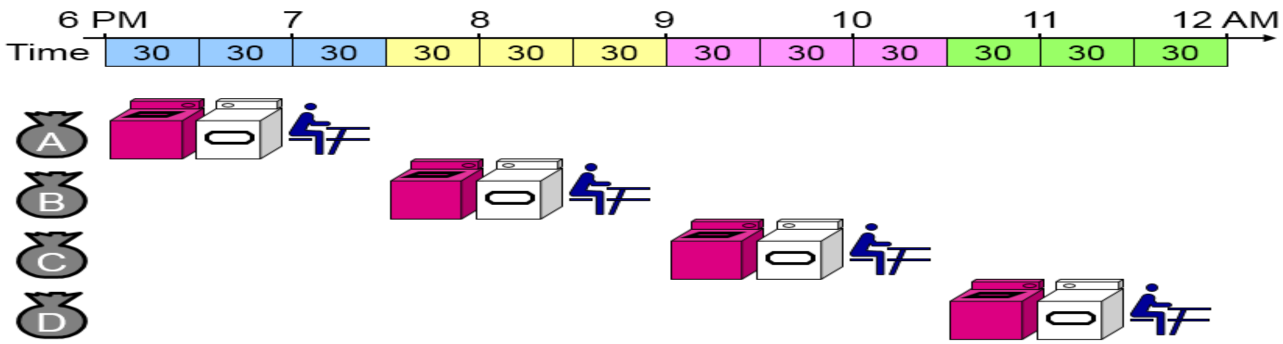
- Laundry Example: Three Stages

1. Wash dirty load of clothes
2. Dry wet clothes
3. Fold and put clothes into drawers

- Each stage takes 30 minutes to complete
- Four loads of clothes to wash, dry, and fold

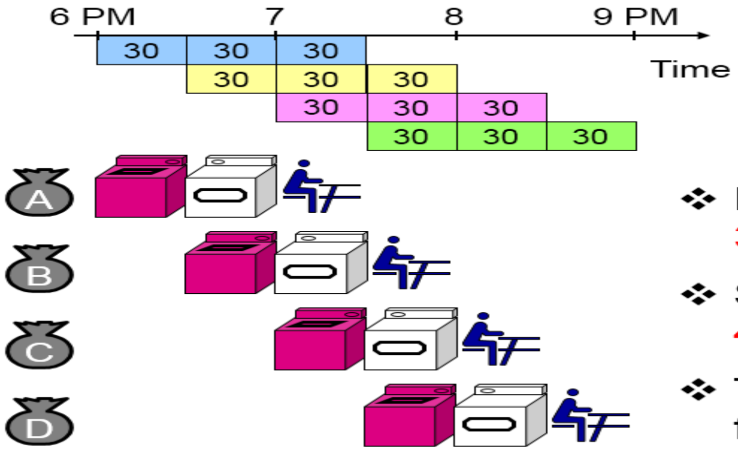


Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- Intuitively, we can use pipelining to speed up laundry

Pipelined Laundry: Start Load ASAP



- Pipelined laundry takes 3 hours for 4 loads
- Speedup factor is 2 for 4 loads
- Time to wash, dry, and fold one load is still the same (90 minutes)

Principles of Pipelined Implementation

- **Break instructions across multiple clock cycles (five, in this case). Design a separate stage for the execution performed during each clock cycle.**

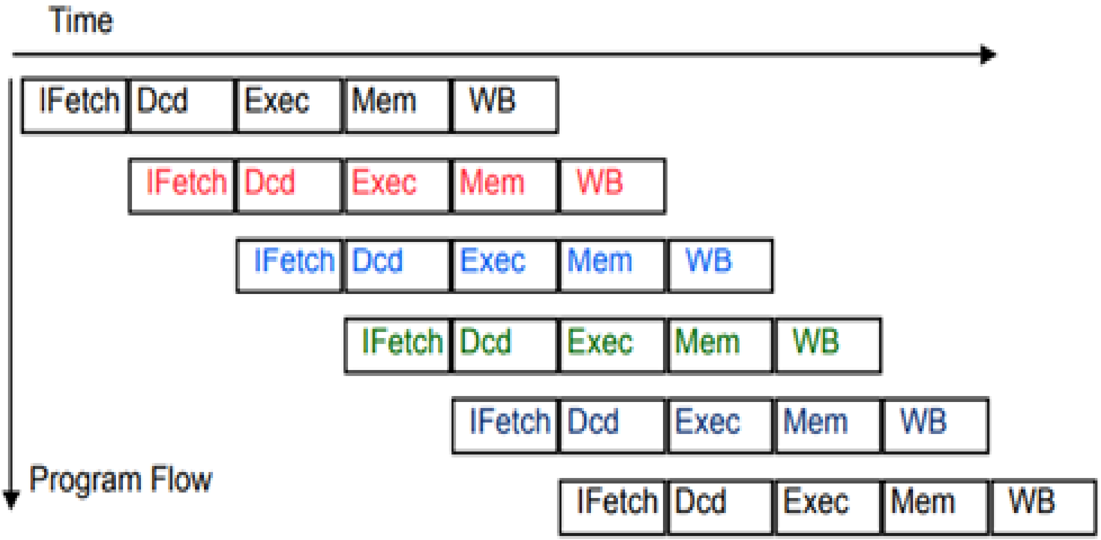
a. **Instruction Fetch (IF)** – get instruction from memory, increment PC

b. **Instruction Decode (ID)** – translate opcode into control signals and read registers

c. **Execute (EX)** – perform ALU operation, compute jump/branch targets

d. **Memory (MEM)** – access memory if needed

e. **Writeback (WB)** – update register file



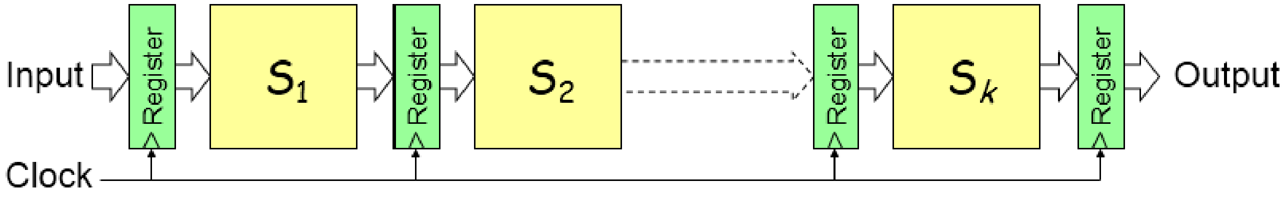
- **Add pipeline registers (flip-flops) to isolate signals between different stages.**

In pipeline system, each segment consists of an **input register** followed by a **combinational circuit**. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.

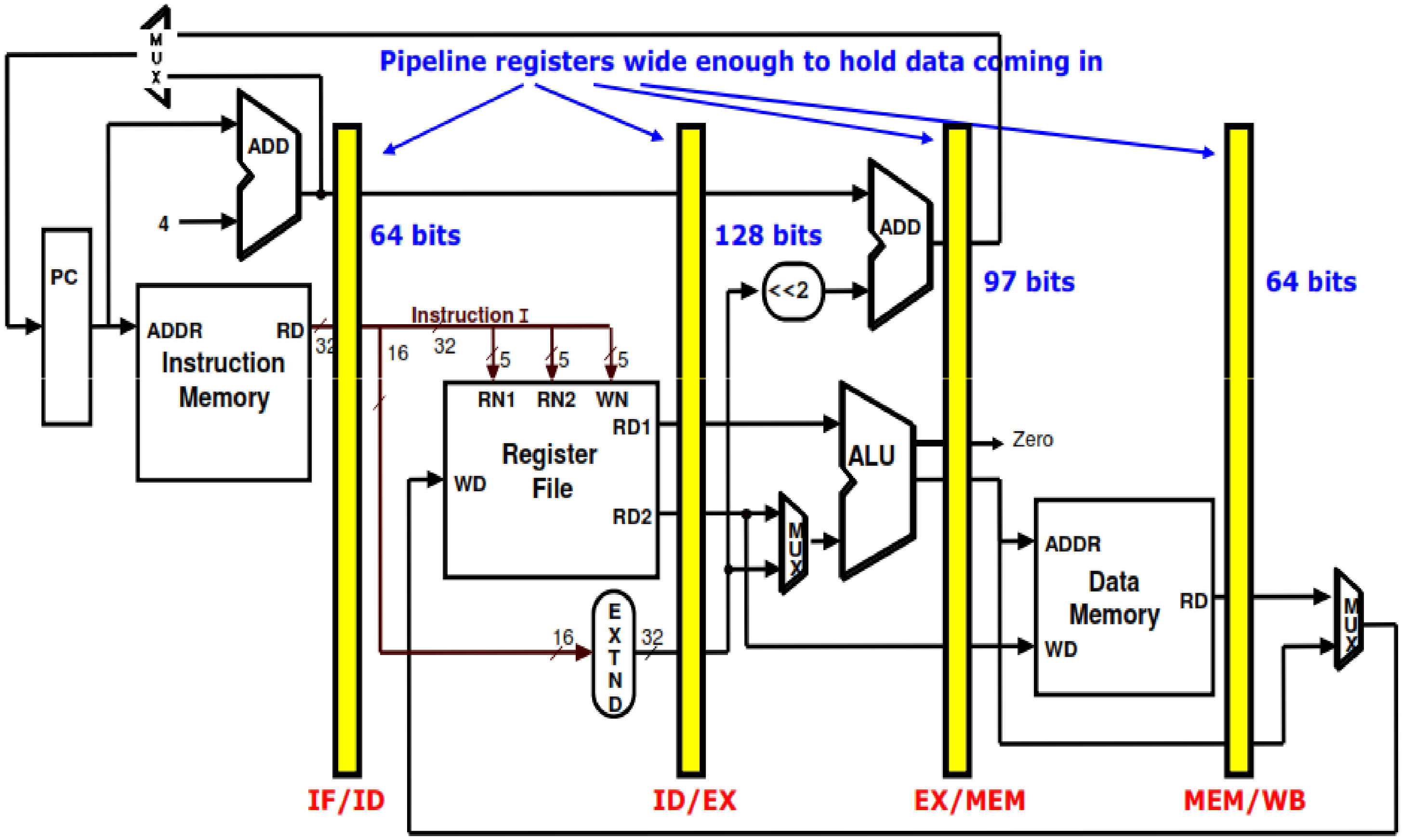
- ❑ **pipeline registers**
- ❑ We can break the execution into multiple cycles, but keep the extra hardware
- ❑ We need *extra* registers to hold data between stages to hold information produced in previous cycle
- ❑ We may be able to start executing a new instruction at each clock cycle (pipeline)

Synchronous Pipeline

- ❖ Uses clocked registers between stages
- ❖ Upon arrival of a clock edge …
  - ✧ All registers hold the results of previous stages simultaneously
- ❖ The pipeline stages are combinational logic circuits
- ❖ It is desirable to have balanced stages
  - ✧ Approximately equal delay in all stages
- ❖ Clock period is determined by the maximum stage delay

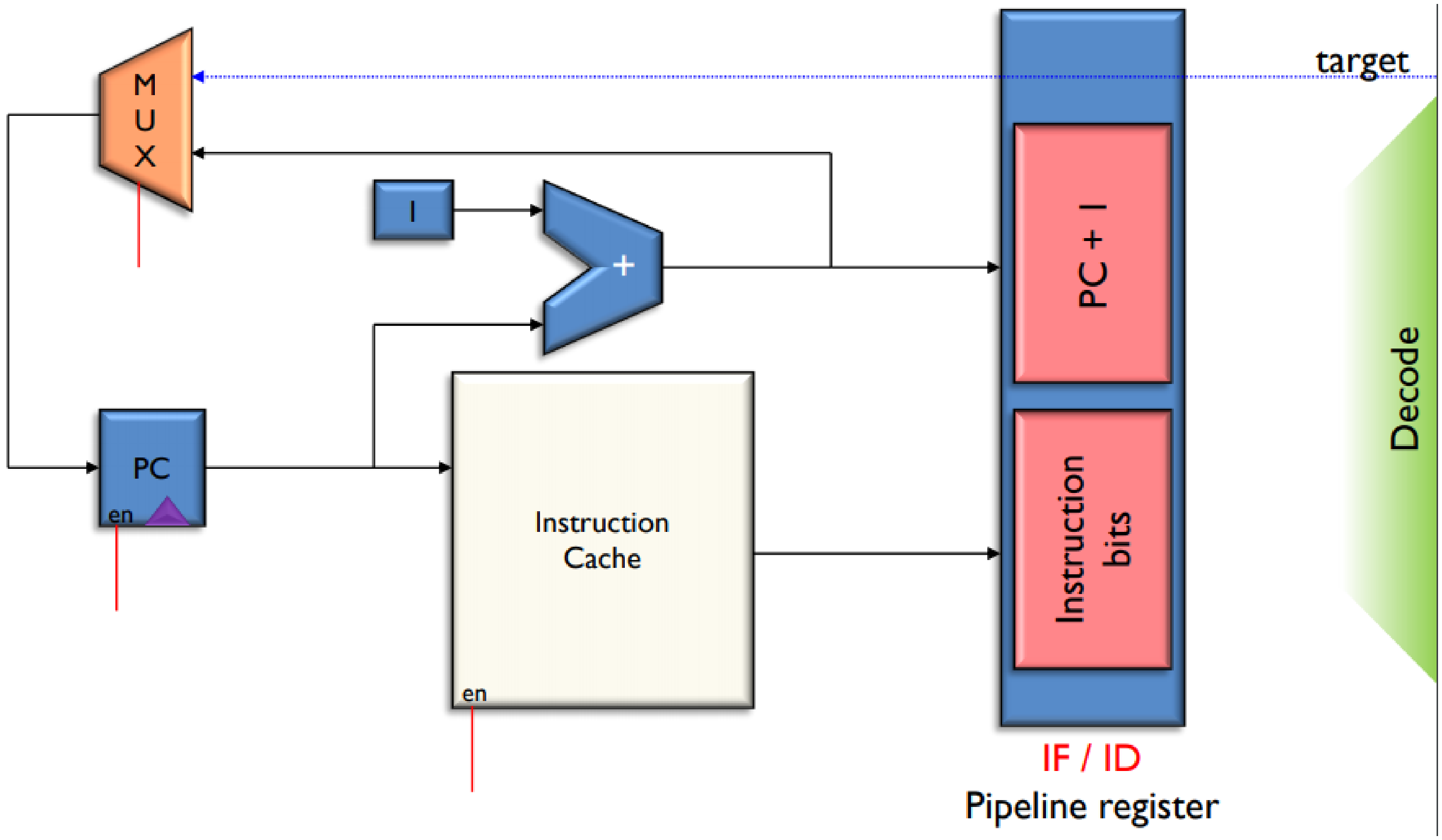


❑ Pipeline registers



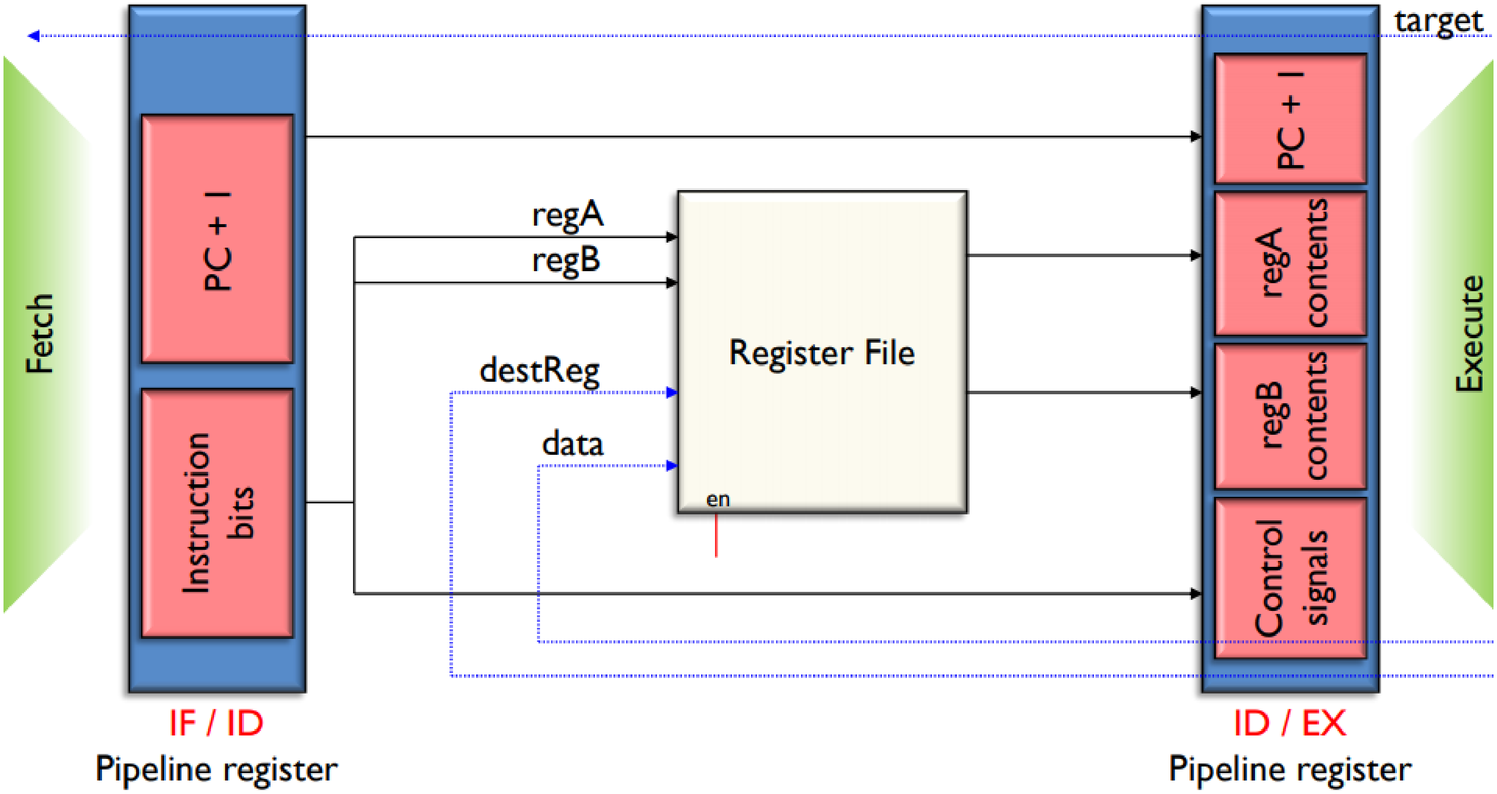
Stage 1: Fetch

- Fetch an instruction from memory every cycle
  - Use PC to index memory
  - Increment PC (assume no branches for now)
- Write state to the pipeline register (IF/ID)
  - The next stage will read this pipeline register



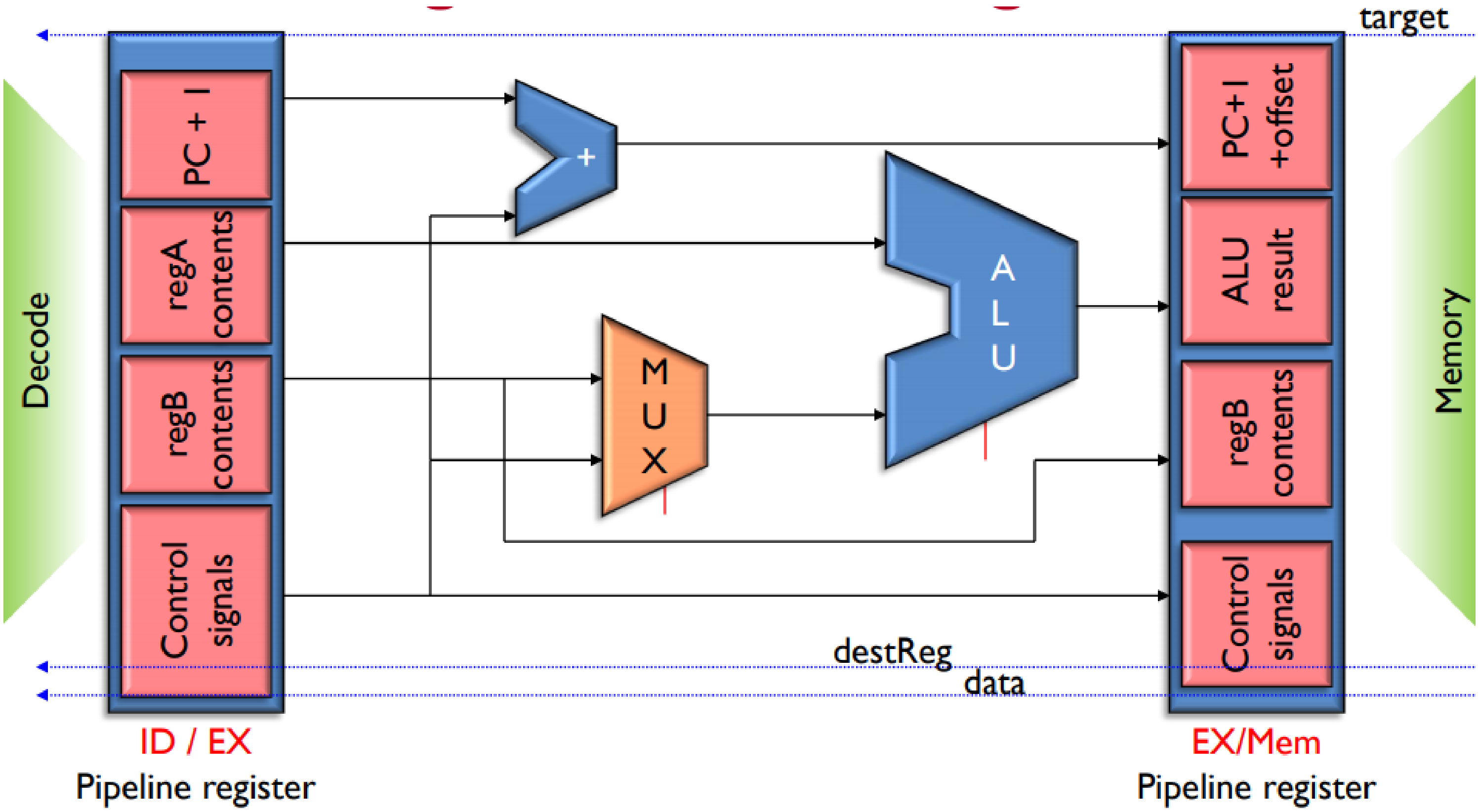
Stage 2: Decode

- Decodes opcode bits
  - Set up Control signals for later stages
- Read input operands from register file
  - Specified by decoded instruction bits
- Write state to the pipeline register (ID/EX)
  - Opcode
  - Register contents
  - PC+1
  - Control signals (from insn) for opcode and destReg



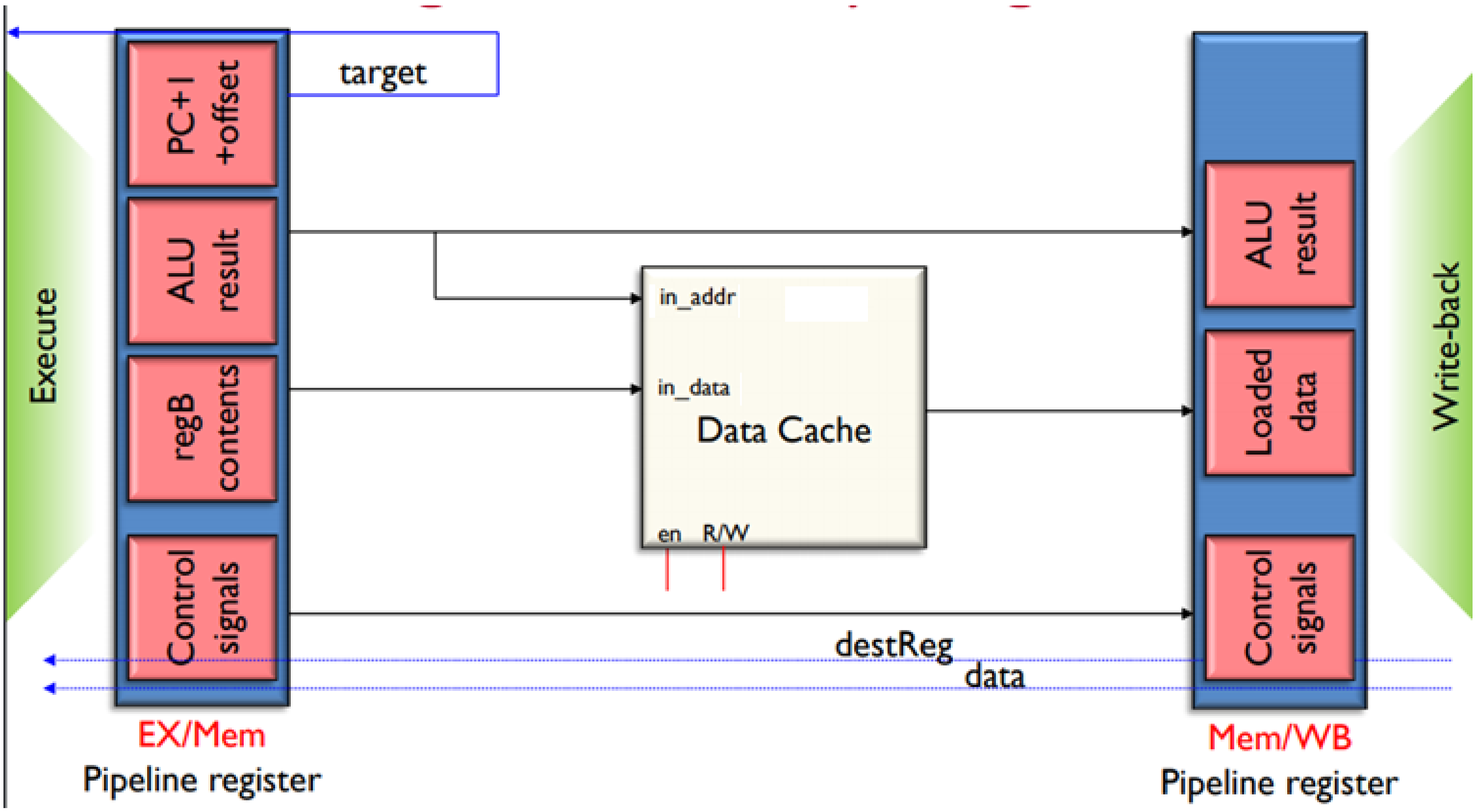
Stage 3: Execute

- Perform ALU operations
  - Calculate result of instruction
    - Control signals select operation
    - Contents of regA used as one input
    - Either regB or constant offset (from insn) used as second input
  - Calculate PC-relative branch target
    - PC+1+(constant offset)
- Write state to the pipeline register (EX/Mem)
  - ALU result, contents of regB, and PC+1+offset
  - Control signals (from insn) for opcode and destReg



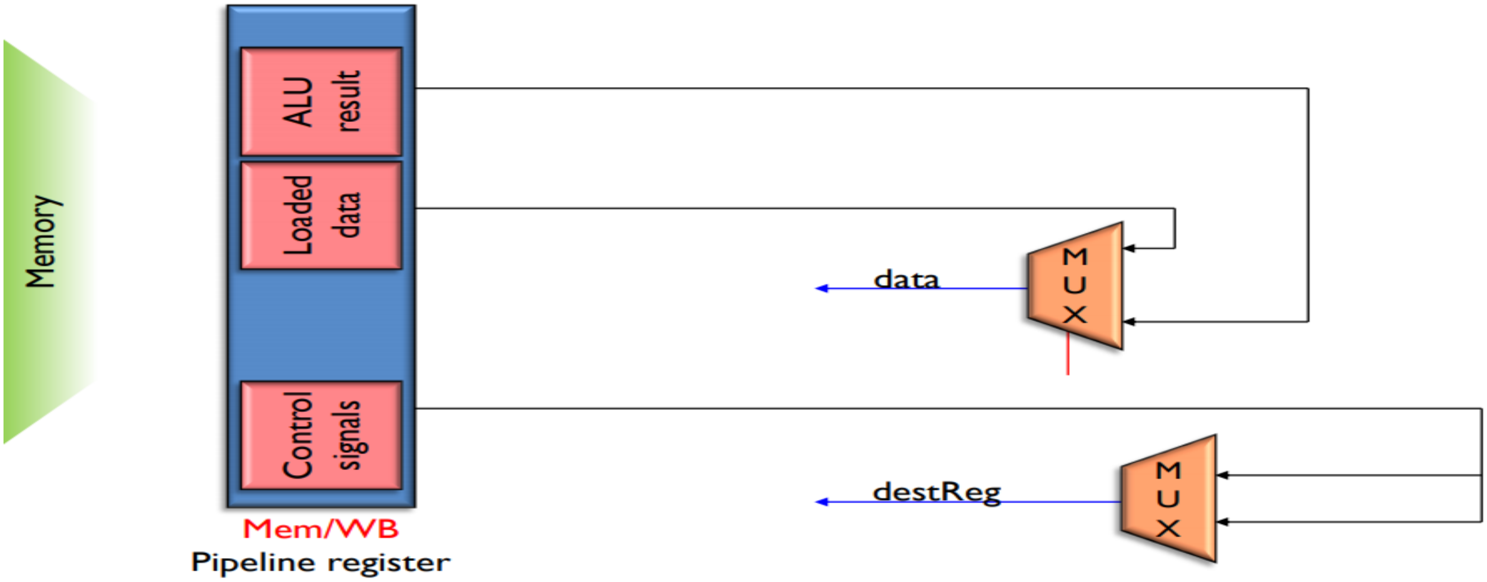
Stage 4: Memory

- Perform data cache access
  - ALU result contains address for LD or ST
  - Opcode bits control R/W and enable signals
- Write state to the pipeline register (Mem/WB)
  - ALU result and Loaded data
  - Control signals (from insn) for opcode and destReg

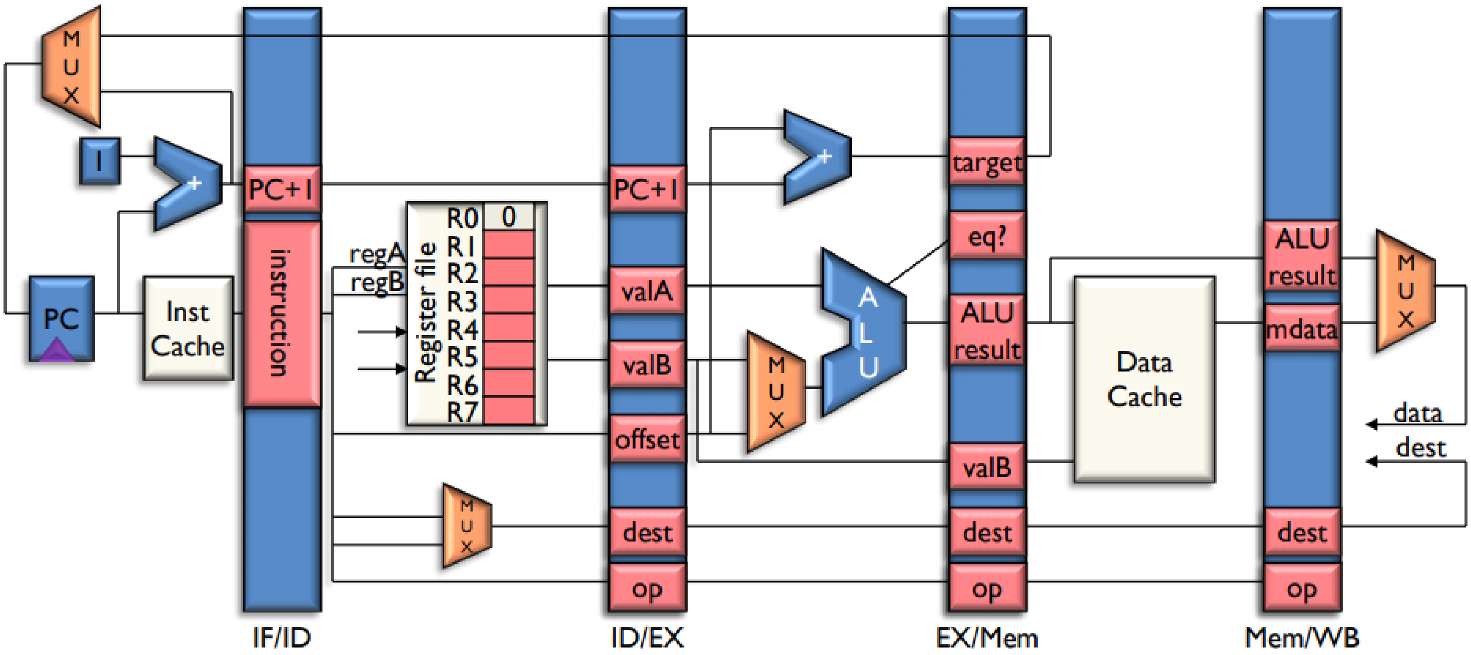


Stage 5: Write-back

- Writing result to register file (if required)
  - Write Loaded data to destReg for LD
  - Write ALU result to destReg for arithmetic insn
  - Opcode bits control register write enable signal



Putting It All Together



A pipeline diagram

Clock cycle

	1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
sub $v0, $a0, $a1		IF	ID	EX	MEM	WB			
and $t1, $t2, $t3			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	MEM	WB	
add $sp, $sp, -4					IF	ID	EX	MEM	WB

- A pipeline diagram shows the execution of a series of instructions.
  - The instruction sequence is shown vertically, from top to bottom.
  - Clock cycles are shown horizontally, from left to right.
  - Each instruction is divided into its component stages. (We show five stages for every instruction, which will make the control unit easier.)
- This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.
  - The "lw" instruction is in its Execute stage.
  - Simultaneously, the "sub" is in its Instruction Decode stage.
  - Also, the "and" instruction is just being fetched.

Pipeline terminology

depth

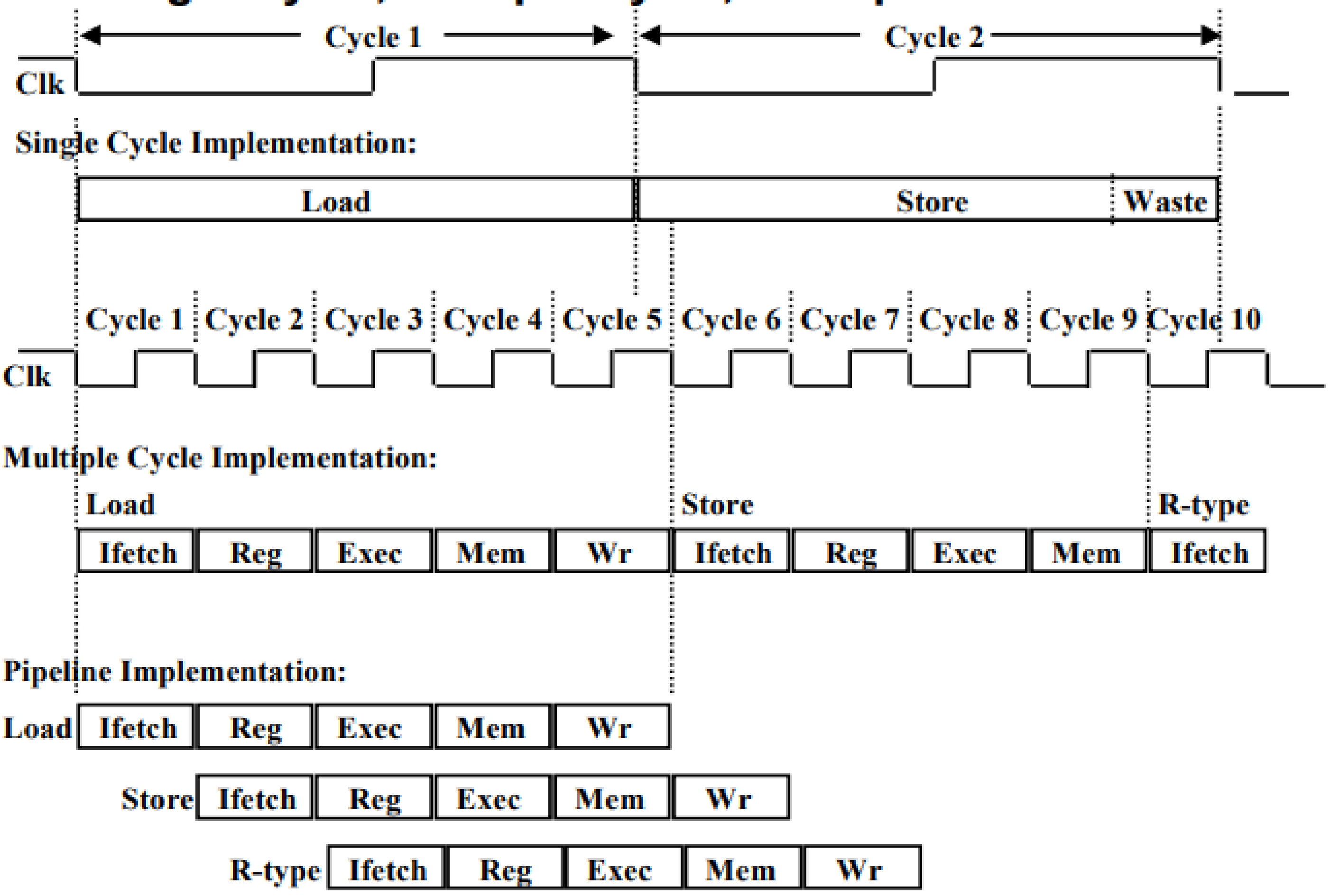
Clock cycle

	1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
sub $v0, $a0, $a1		IF	ID	EX	MEM	WB			
and $t1, $t2, $t3			IF	ID	EX	MEM	WB		
or $s0, $s1, $s2				IF	ID	EX	MEM	WB	
add $sp, $sp, -4					IF	ID	EX	MEM	WB

filling — full — emptying

- The pipeline depth is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is filling, since there are unused functional units.
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9, the pipeline is emptying.

Single Cycle, Multiple Cycle, vs. Pipeline



If there are k stages, and each stage takes t time units, then the time needed to execute N instructions is

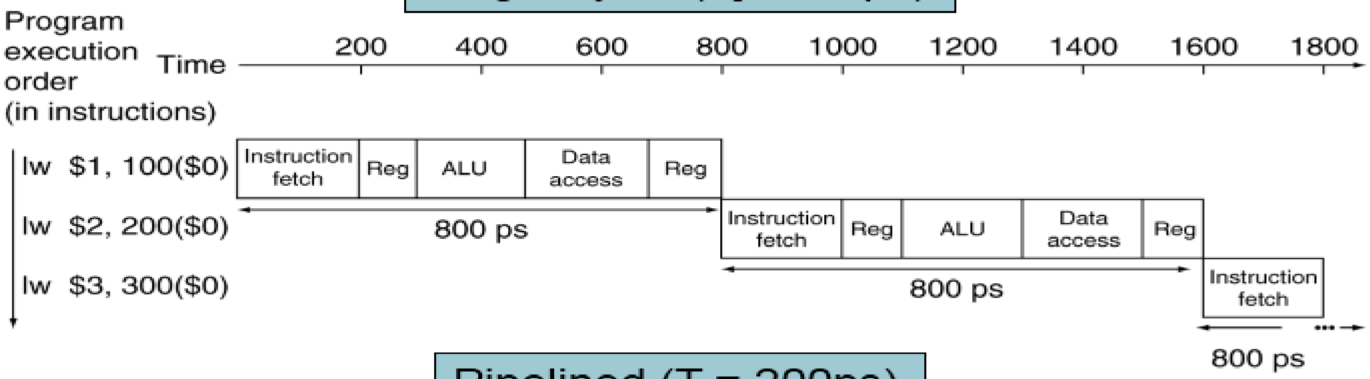
$$k.t + (N-1).t$$

Pipeline Performance

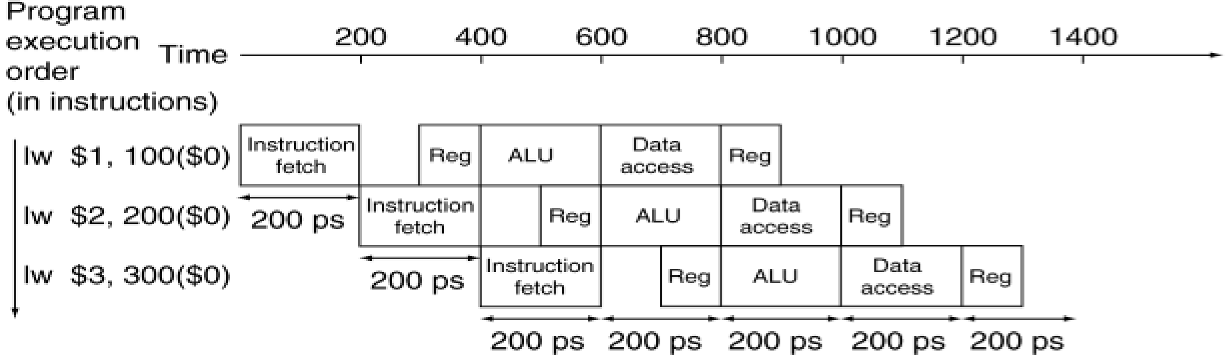
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Single-cycle (T_C= 800ps)



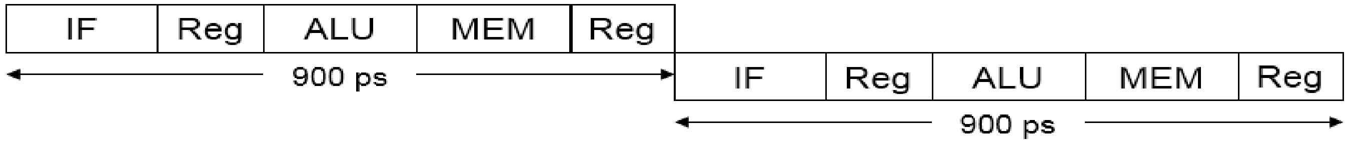
Pipelined (T_C= 200ps)



Single-Cycle vs Pipelined Performance

- Consider a 5-stage instruction execution in which …
  - Instruction fetch = ALU operation = Data memory access = 200 ps
  - Register read = register write = 150 ps
- What is the single-cycle non-pipelined time?
- What is the pipelined cycle time?
- What is the speedup factor for pipelined execution?
- Solution

  Non-pipelined cycle = 200+150+200+200+150 = 900 ps



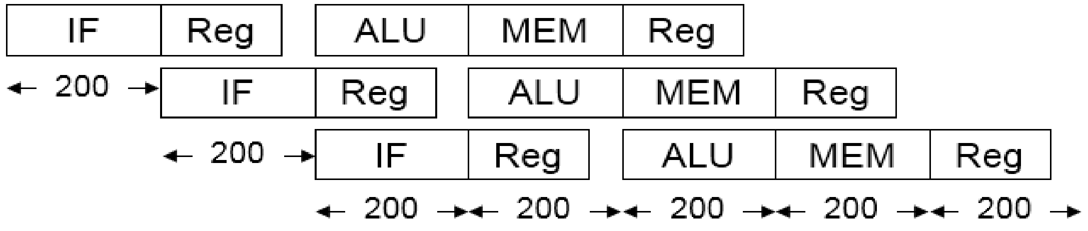
IF | Reg | ALU | MEM | Reg

900 ps

IF | Reg | ALU | MEM | Reg

900 ps

- Pipelined cycle time = max(200, 150) = 200 ps



IF | Reg | ALU | MEM | Reg

200 | IF | Reg | ALU | MEM | Reg

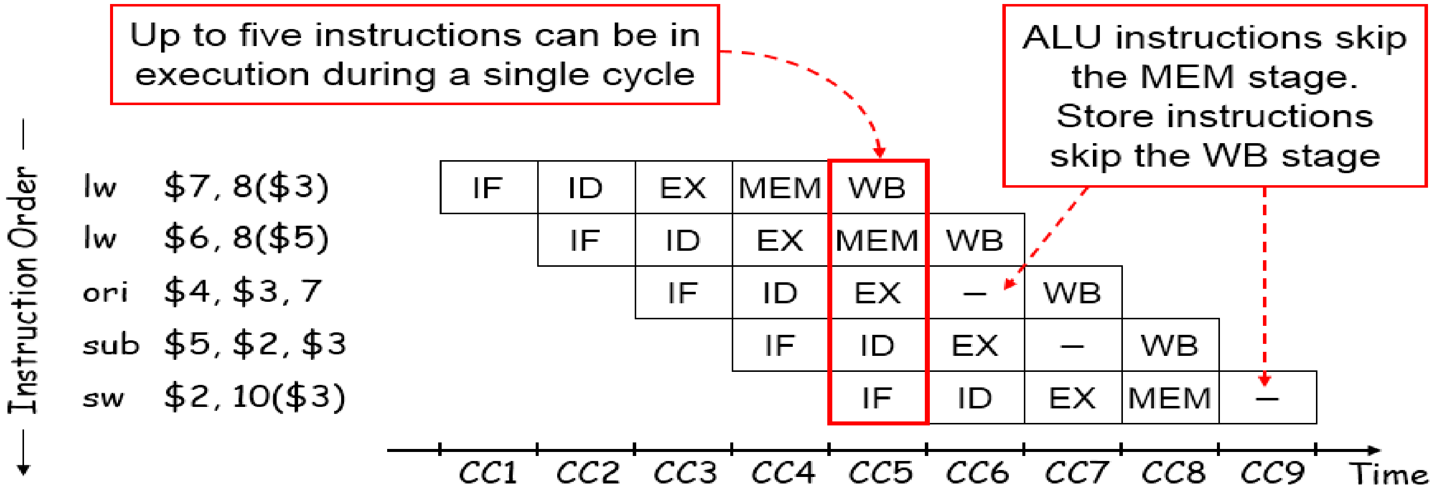
200 | IF | Reg | ALU | MEM | Reg

200 | 200 | 200 | 200 | 200

- CPI for pipelined execution = 1
  - One instruction completes each cycle (ignoring pipeline fill)
- Speedup of pipelined execution = 900 ps / 200 ps = 4.5
  - Instruction count and CPI are equal in both cases
- Speedup factor is less than 5 (number of pipeline stage)
  - Because the pipeline stages are not balanced

Instruction-Time Diagram

- Diagram shows:
  - Which instruction occupies what stage at each clock cycle
- Instruction execution is pipelined over the 5 stages



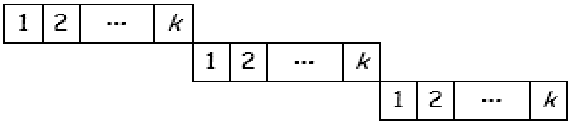
Up to five instructions can be in execution during a single cycle

ALU instructions skip the MEM stage. Store instructions skip the WB stage

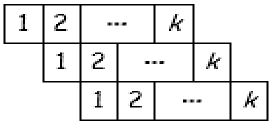
Instruction Order	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
lw $7, 8($3)	IF	ID	EX	MEM	WB				
lw $6, 8($5)		IF	ID	EX	MEM	WB			
ori $4, $3, 7			IF	ID	EX	–	WB		
sub $5, $2, $3				IF	ID	EX	–	WB	
sw $2, 10($3)					IF	ID	EX	MEM	–

Serial Execution versus Pipelining

- Consider a task that can be divided into *k* subtasks
  - The *k* subtasks are executed on *k* different stages
  - Each subtask requires one time unit
  - The total execution time of the task is *k* time units
- Pipelining is to overlap the execution
  - The *k* stages work in parallel on *k* different tasks
  - Tasks enter/leave pipeline at the rate of one task per time unit

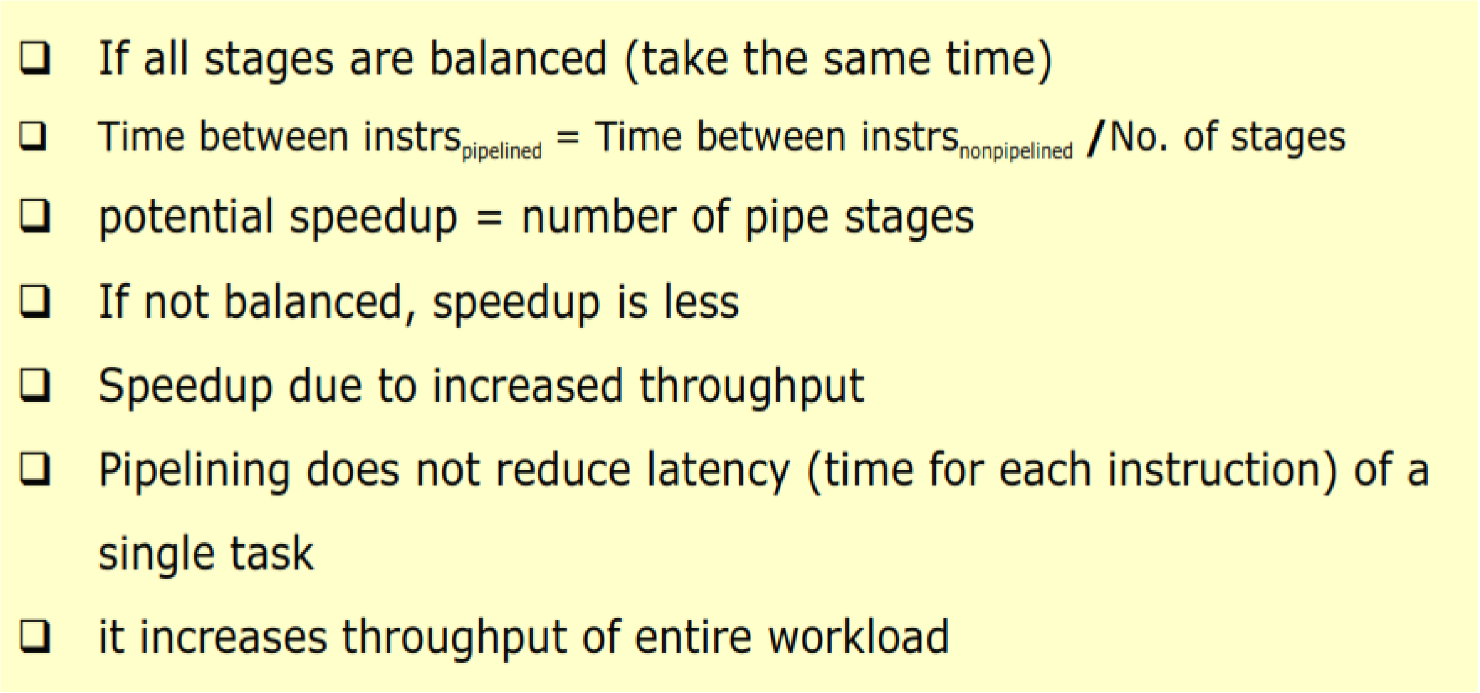


Without Pipelining
One completion every *k* time units

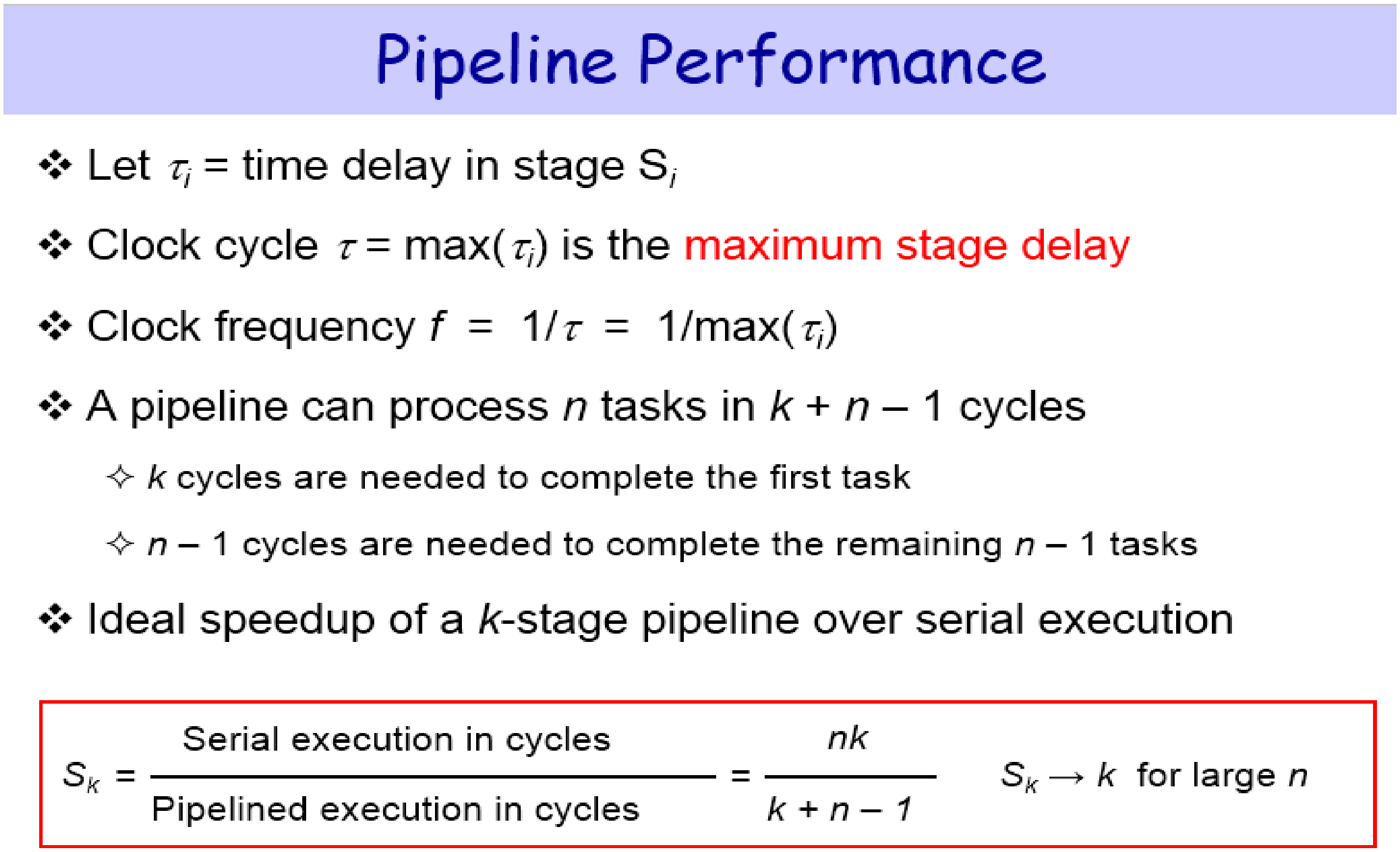


With Pipelining
One completion every 1 time unit

Pipeline Speedup



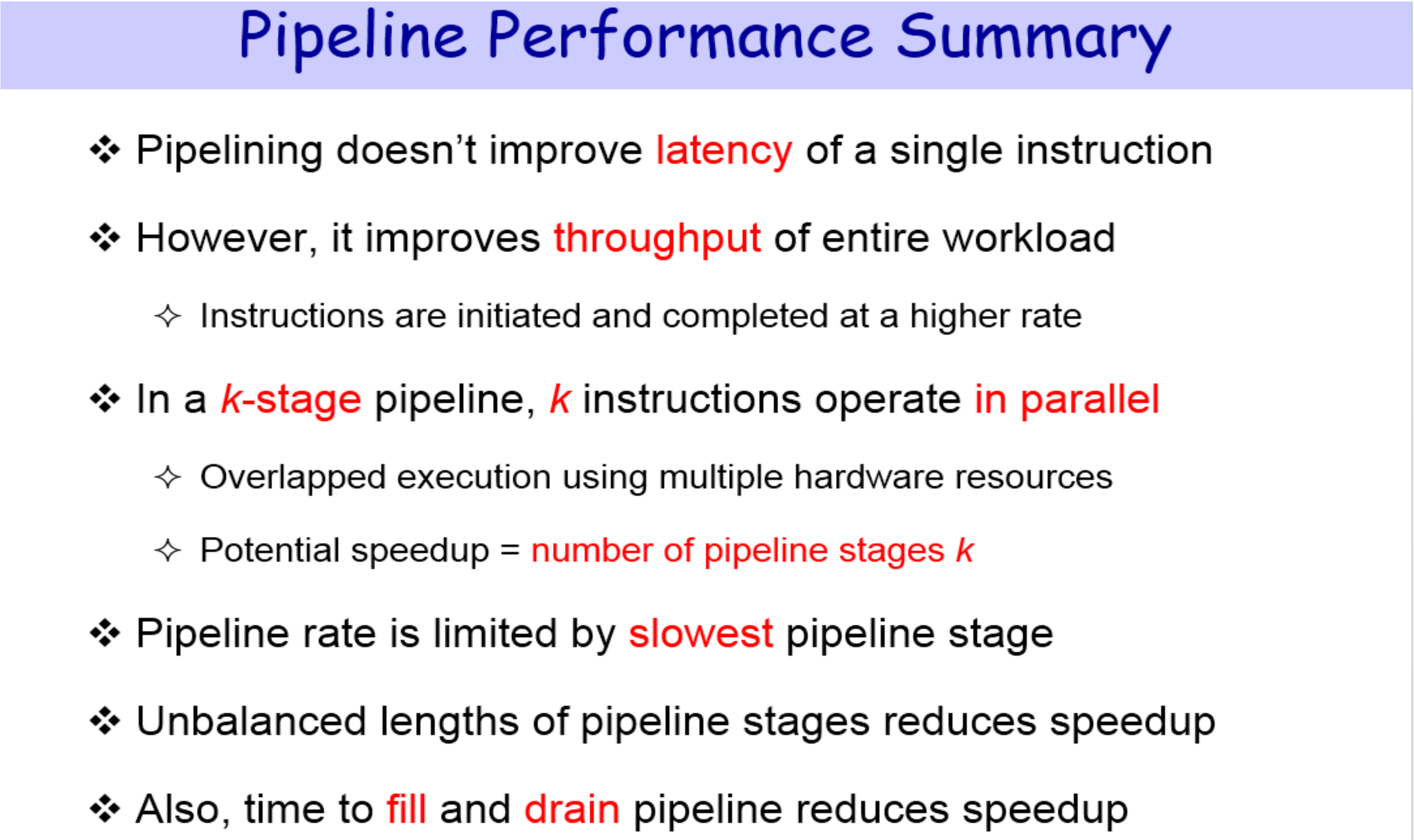
- ❑ If all stages are balanced (take the same time)
- ❑ Time between $\text{instrs}_{\text{pipelined}}$ = Time between $\text{instrs}_{\text{nonpipelined}}$ /No. of stages
- ❑ potential speedup = number of pipe stages
- ❑ If not balanced, speedup is less
- ❑ Speedup due to increased throughput
- ❑ Pipelining does not reduce latency (time for each instruction) of a single task
- ❑ it increases throughput of entire workload



Pipeline Performance

- ❖ Let τ_i = time delay in stage S_i
- ❖ Clock cycle $\tau = \max(\tau_i)$ is the maximum stage delay
- ❖ Clock frequency $f = 1/\tau = 1/\max(\tau_i)$
- ❖ A pipeline can process n tasks in $k + n - 1$ cycles
  - ✧ k cycles are needed to complete the first task
  - ✧ $n - 1$ cycles are needed to complete the remaining $n - 1$ tasks
- ❖ Ideal speedup of a k-stage pipeline over serial execution

$$S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k + n - 1} \qquad S_k \to k \text{ for large } n$$



Pipeline Performance Summary

- ❖ Pipelining doesn’t improve latency of a single instruction
- ❖ However, it improves throughput of entire workload
  - ✧ Instructions are initiated and completed at a higher rate
- ❖ In a *k*-stage pipeline, *k* instructions operate in parallel
  - ✧ Overlapped execution using multiple hardware resources
  - ✧ Potential speedup = number of pipeline stages *k*
- ❖ Pipeline rate is limited by slowest pipeline stage
- ❖ Unbalanced lengths of pipeline stages reduces speedup
- ❖ Also, time to fill and drain pipeline reduces speedup

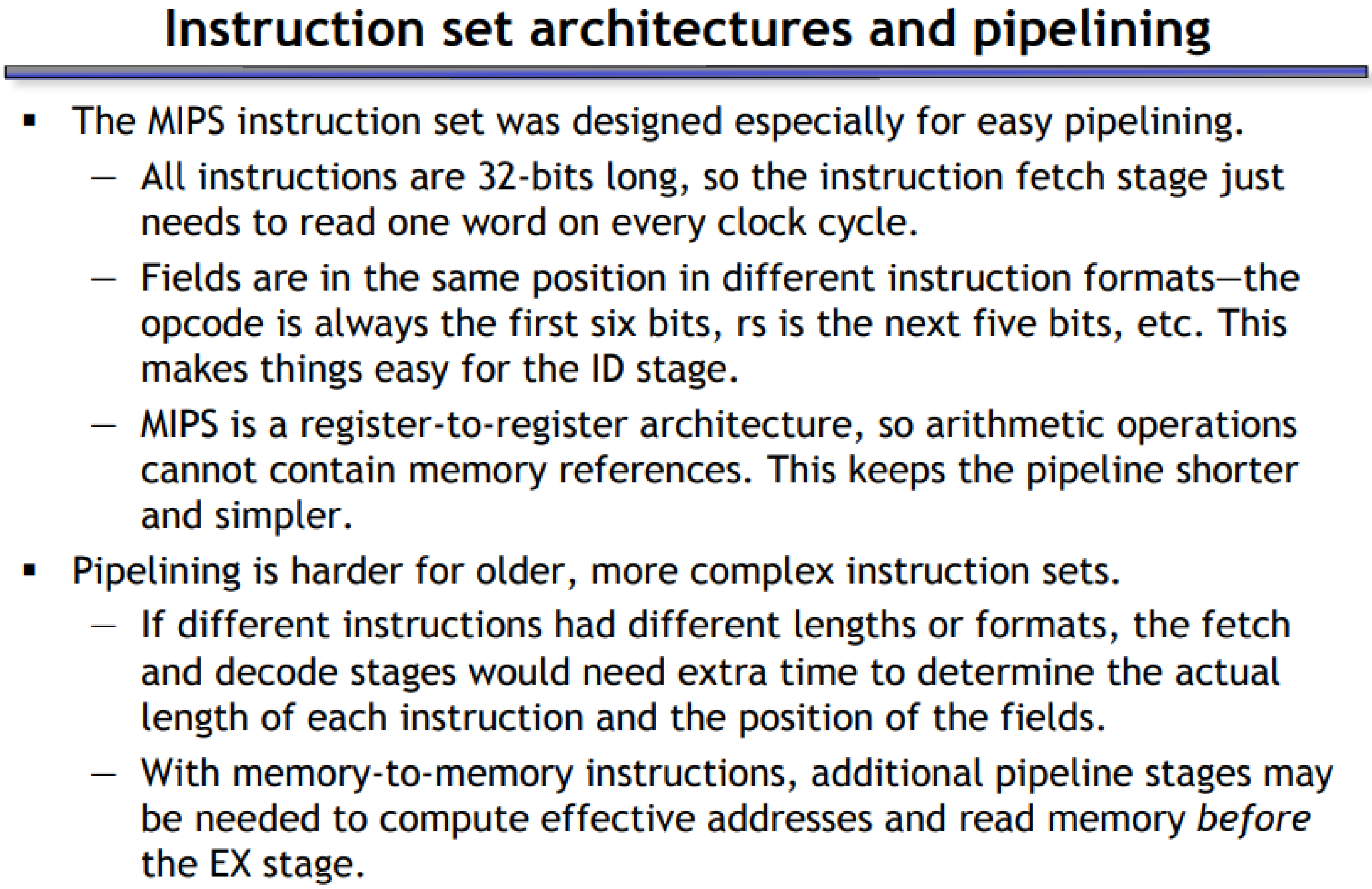
Design Instruction Sets for Pipelining

First, all MIPS instructions are the **same length (32-bits)**. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage.

Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the **second stage** can begin **reading the register** file at the same time that the hardware is determining **what type of instruction** was fetched.

Third, memory operands only appear in **loads or stores** in MIPS. This restriction means we can use the execute stage **(3rd stage)** to calculate the memory address and then access memory in the following stage **(4th stage)**.

Fourth, operands must be aligned in memory. Memory access takes only one cycle.



Instruction set architectures and pipelining

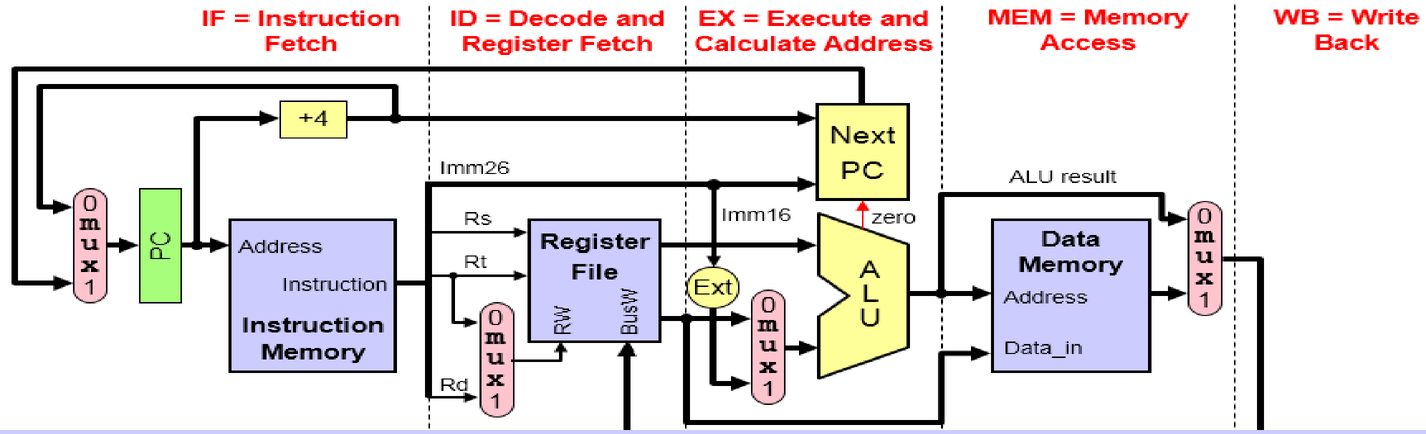
- The MIPS instruction set was designed especially for easy pipelining.
  - All instructions are 32-bits long, so the instruction fetch stage just needs to read one word on every clock cycle.
  - Fields are in the same position in different instruction formats—the opcode is always the first six bits, rs is the next five bits, etc. This makes things easy for the ID stage.
  - MIPS is a register-to-register architecture, so arithmetic operations cannot contain memory references. This keeps the pipeline shorter and simpler.
- Pipelining is harder for older, more complex instruction sets.
  - If different instructions had different lengths or formats, the fetch and decode stages would need extra time to determine the actual length of each instruction and the position of the fields.
  - With memory-to-memory instructions, additional pipeline stages may be needed to compute effective addresses and read memory *before* the EX stage.

Pipelined Datapath and Control

Single-Cycle Datapath

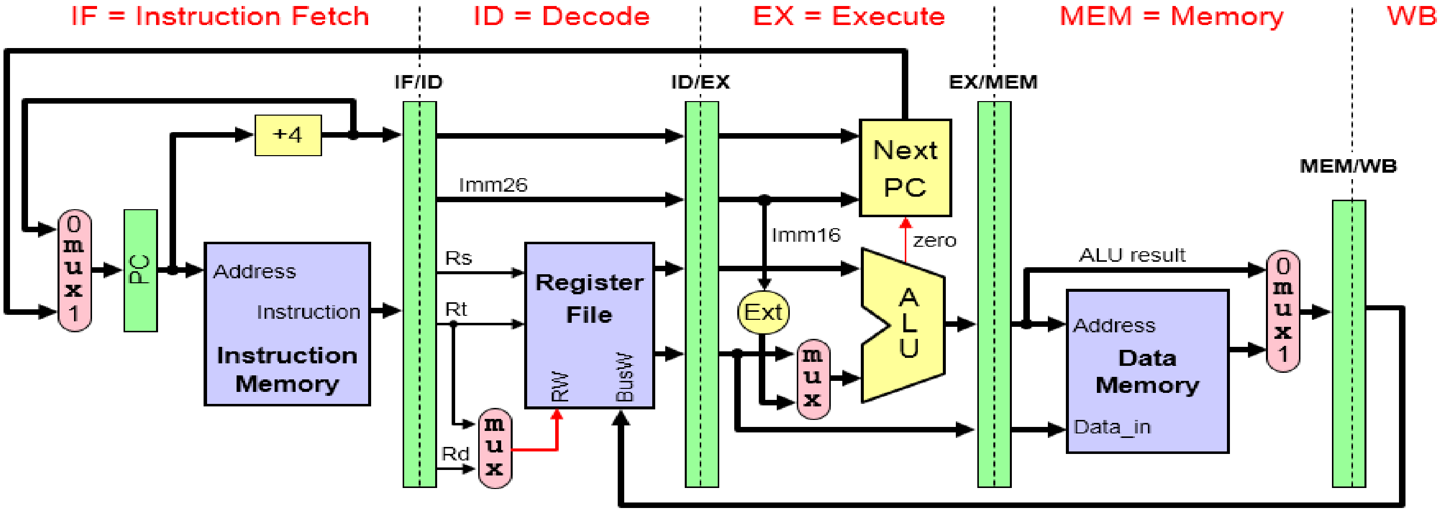
- Shown below is the single-cycle datapath
- How to pipeline this single-cycle datapath?

  Answer: Introduce registers at the end of each stage

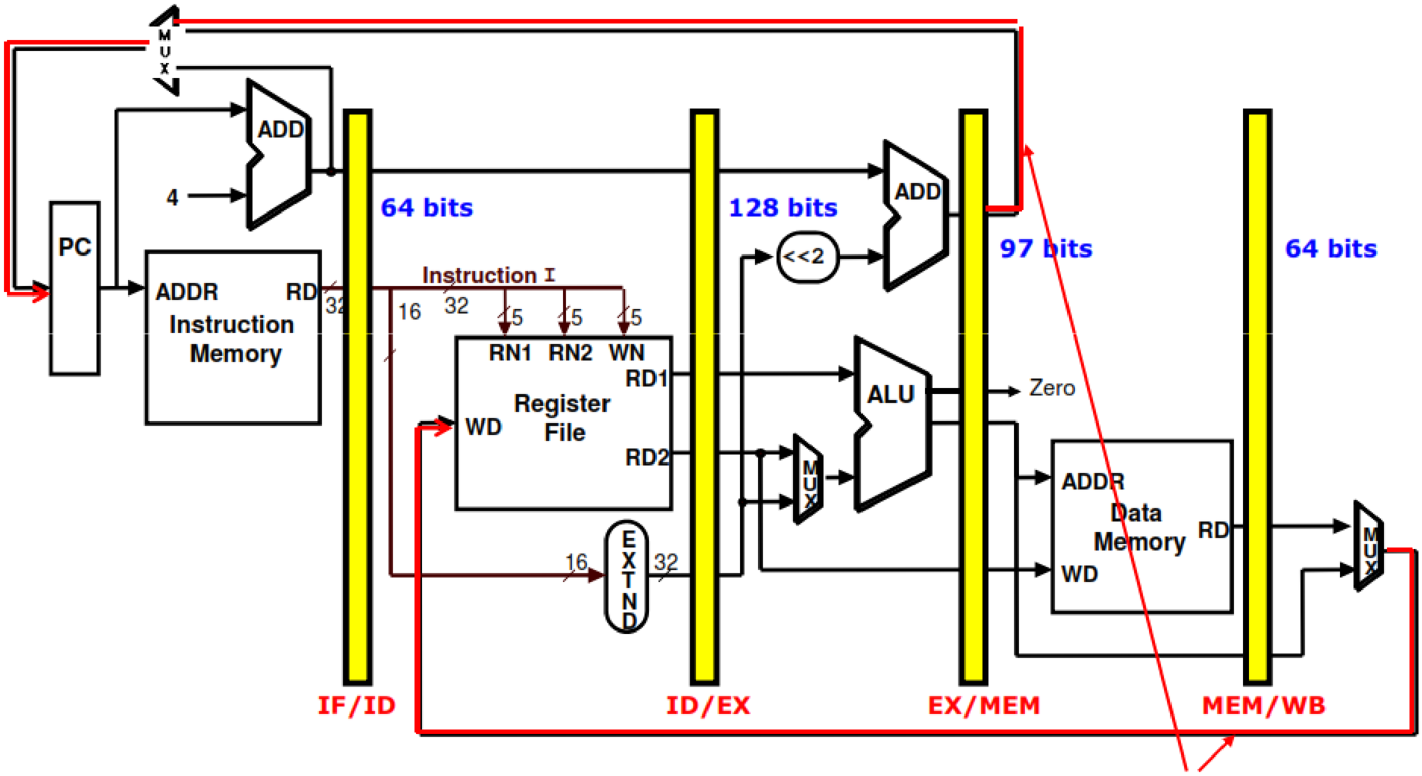


Pipelined Datapath

- Pipeline registers, in **green**, separate each pipeline stage and hold information produced in previous cycle
- The registers must be wide enough to store all the data corresponding to the lines that go through them.
- Pipeline registers are labeled by the stages they separate
- Is there a problem with the register destination address?



❑ Right-to-left flow leads to hazards



Only data flowing right to left may cause hazard

- **Pipeline registers propagate data and control values to later stages.**

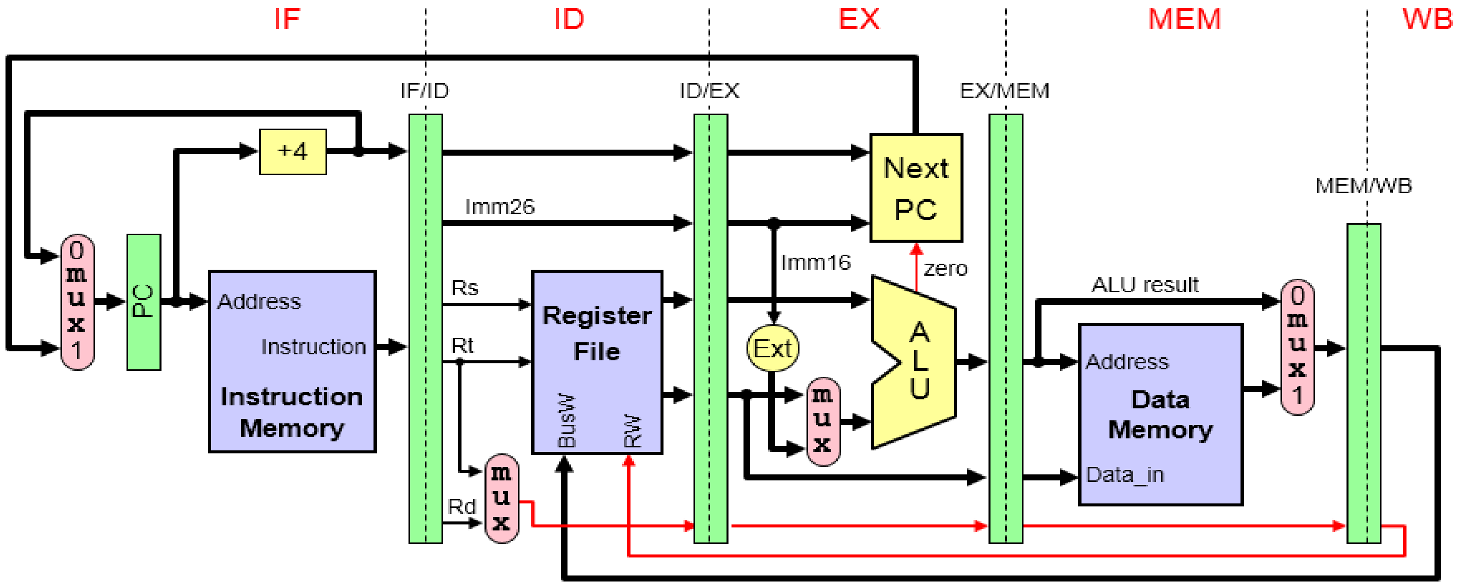
- **Each step of the instruction can be mapped onto the datapath from left to right.**

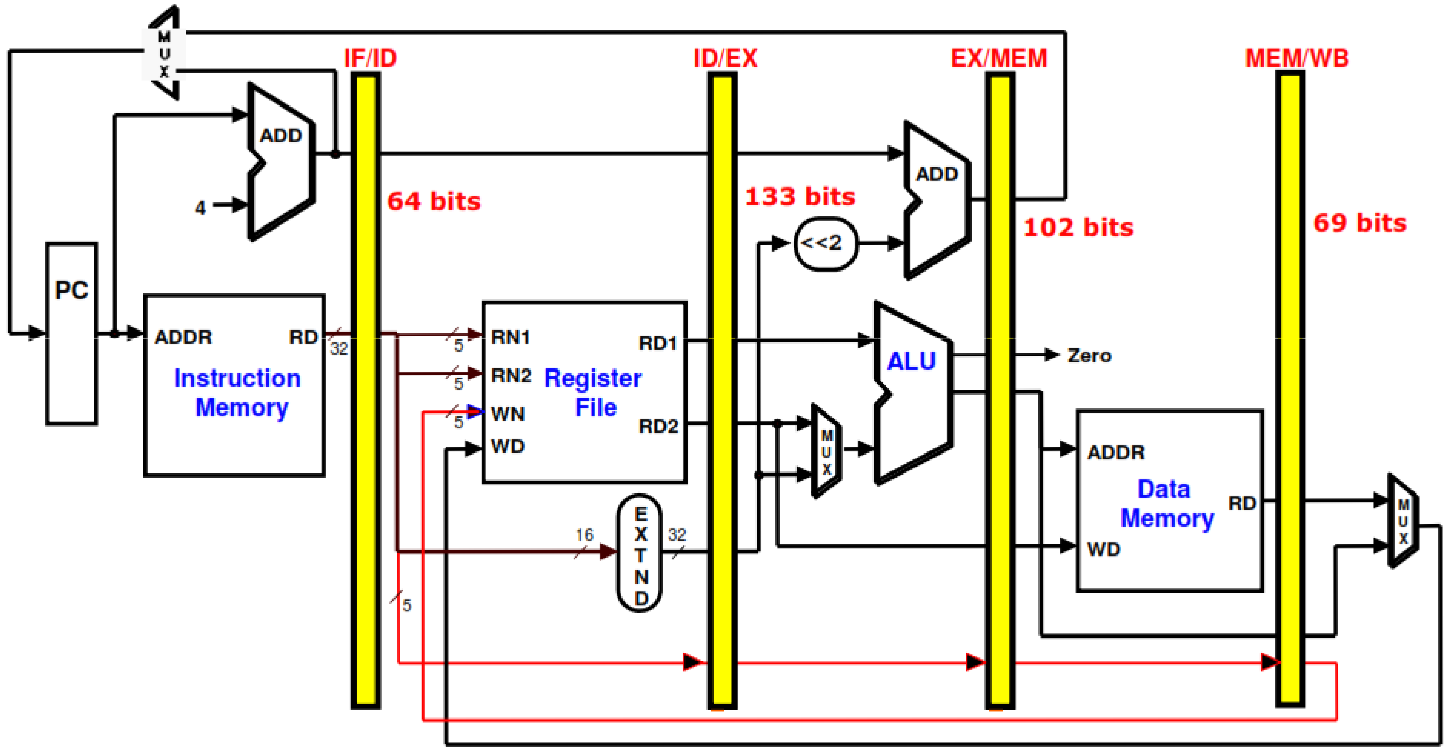
- **There are two exceptions to this left -to-right flow of instructions:**
  - The only exceptions are the update of the PC (***choosing between the incremented PC and the branch address***).
  - The write-back step, which sends either ***the ALU result or the data from memory*** to the left to be written into the register file.

- **Data flowing from right to left does not affect the current instruction; these reverse data movements influence only later instructions in the pipeline**

Corrected Pipelined Datapath

- Destination register number should come from MEM/WB
  - Along with the data during the written back stage
- Destination register number is passed from ID to WB stage



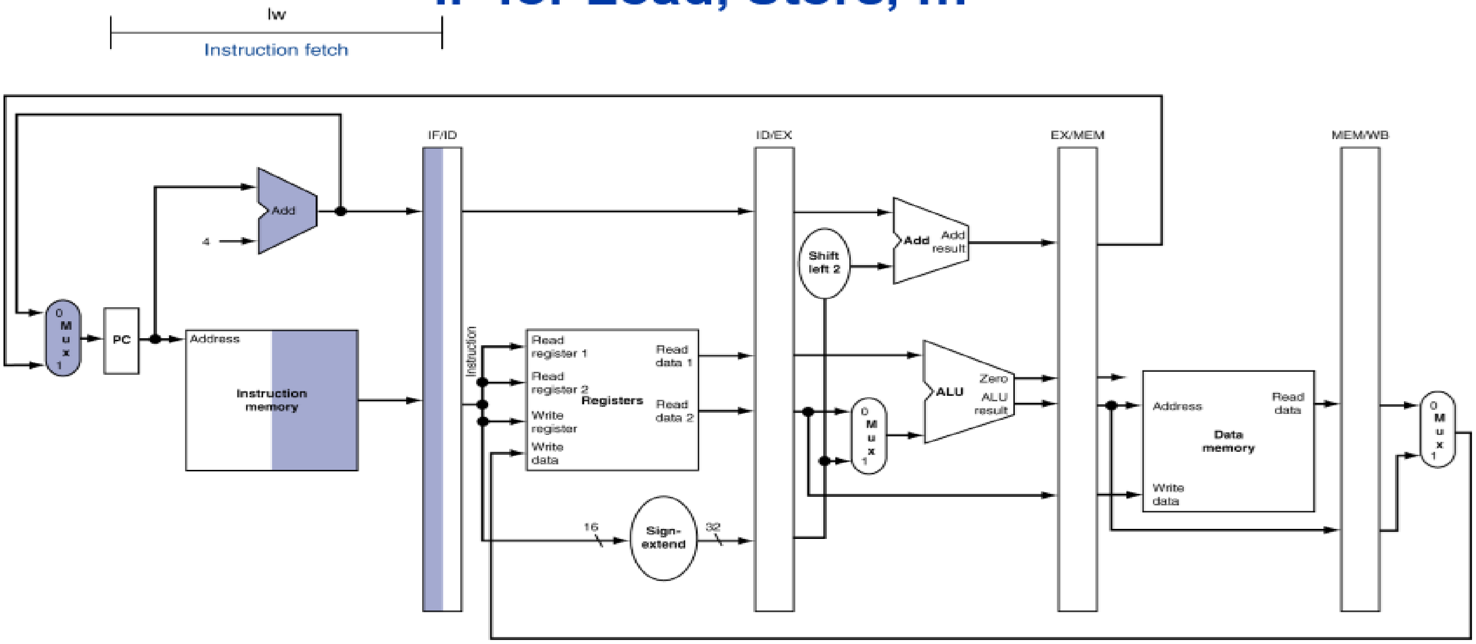


Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits

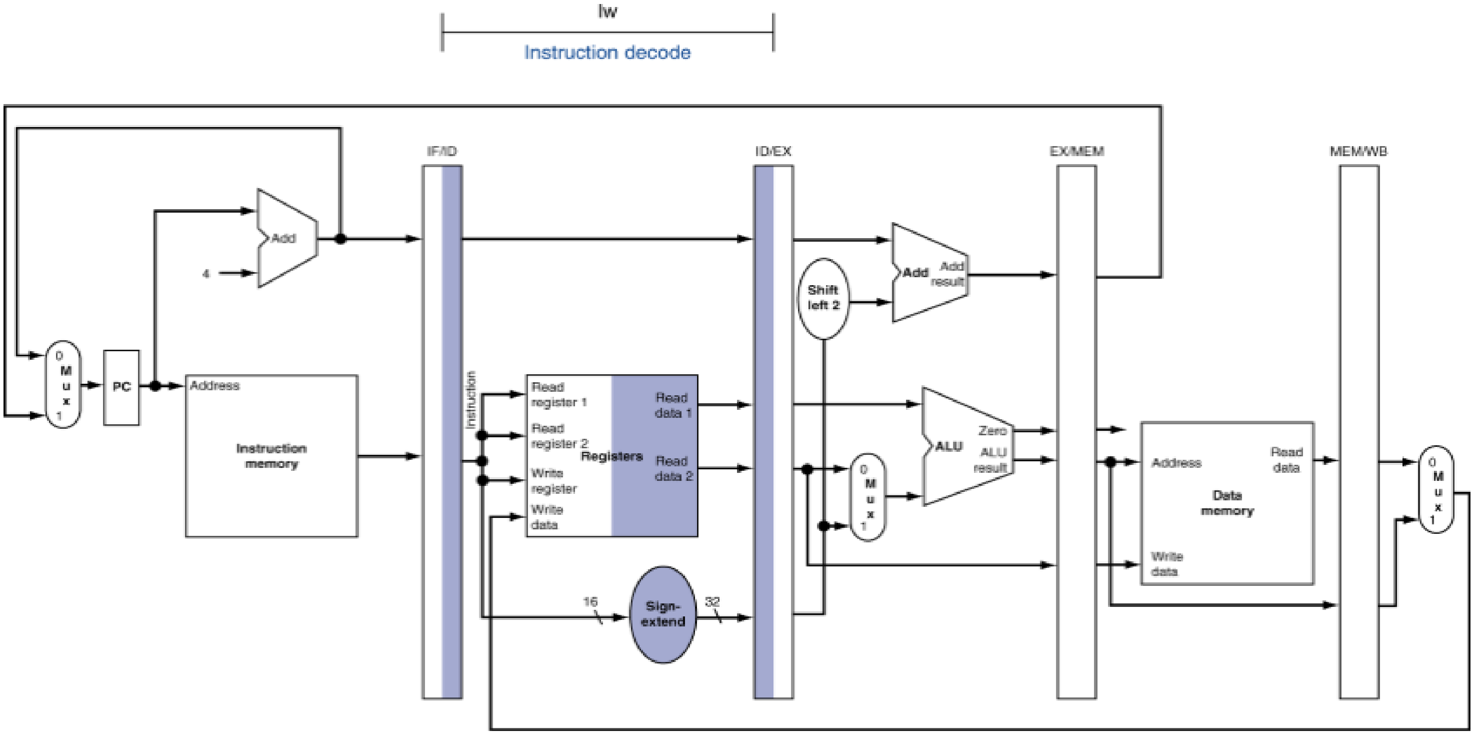
Pipeline Operation

Cycle-by-cycle flow of instructions through the pipelined datapath for load & store

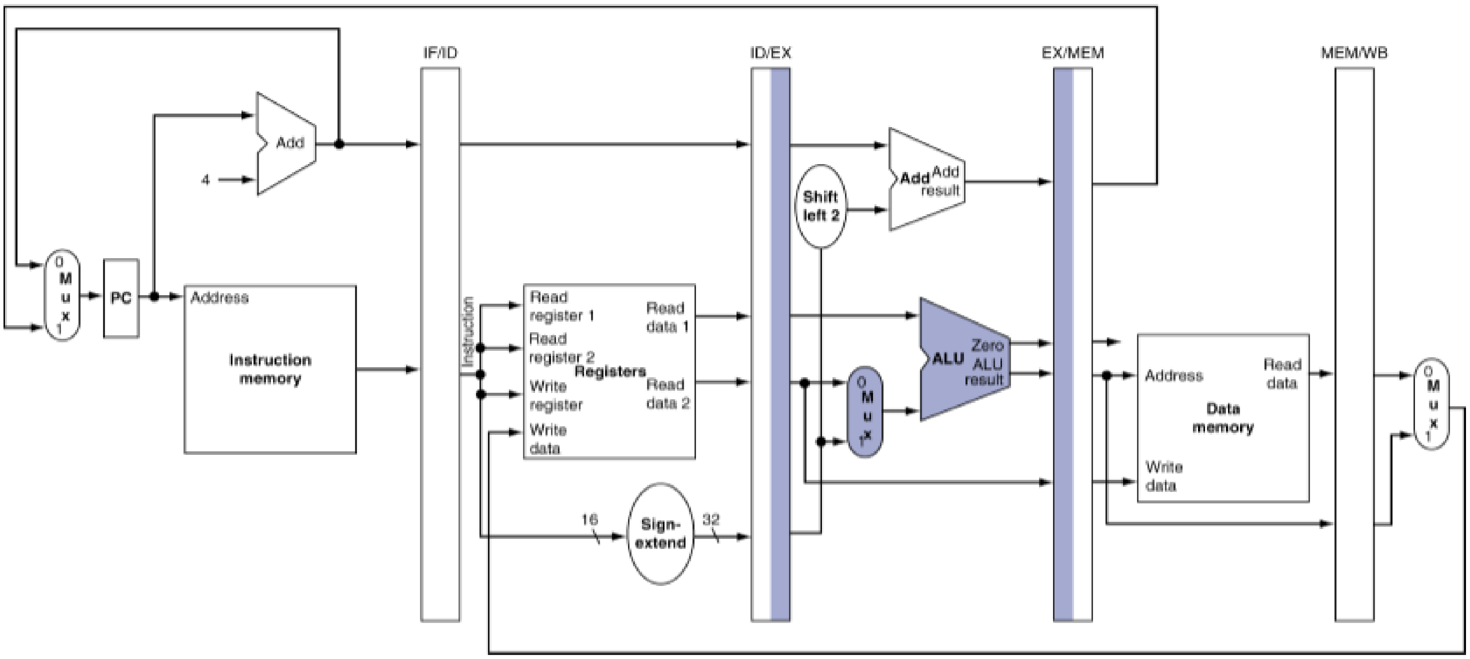
IF for Load, Store, …



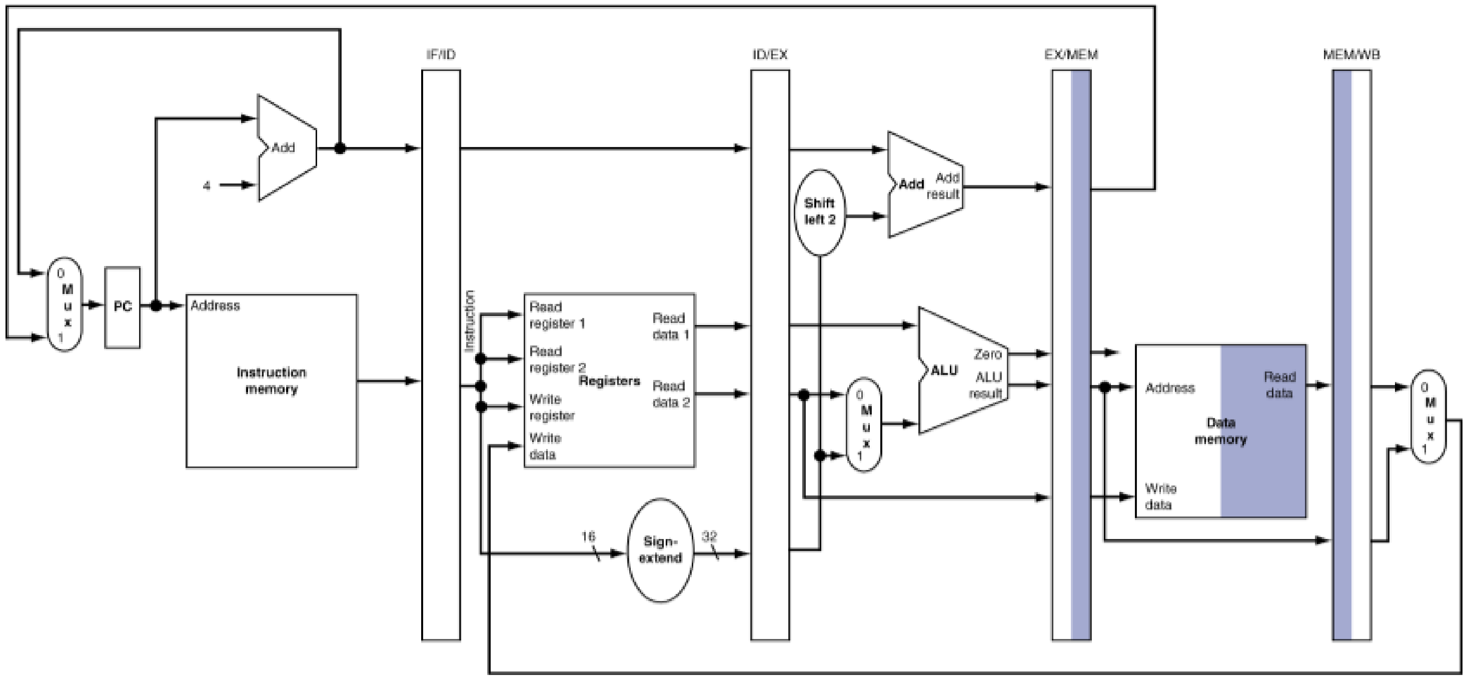
ID for Load, Store, …



EX for Load



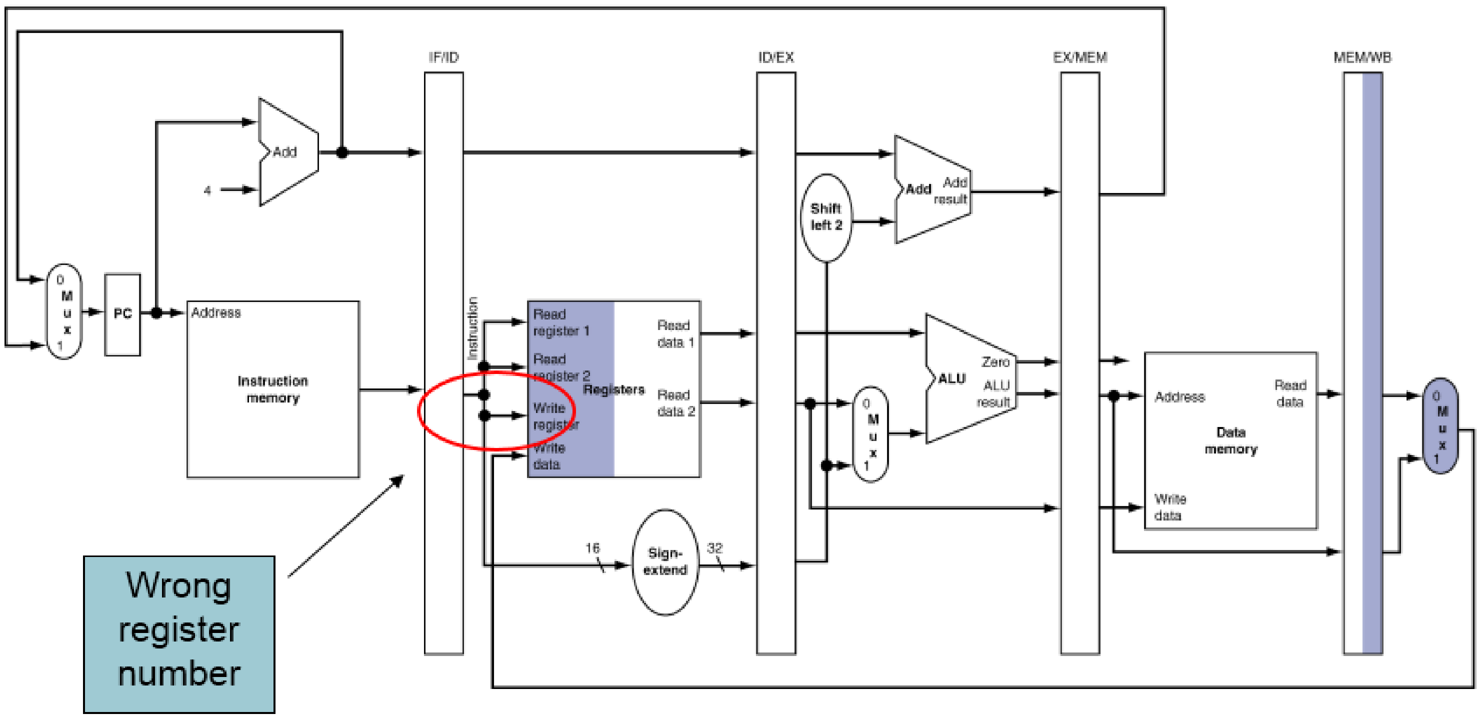
MEM for Load



WB for Load

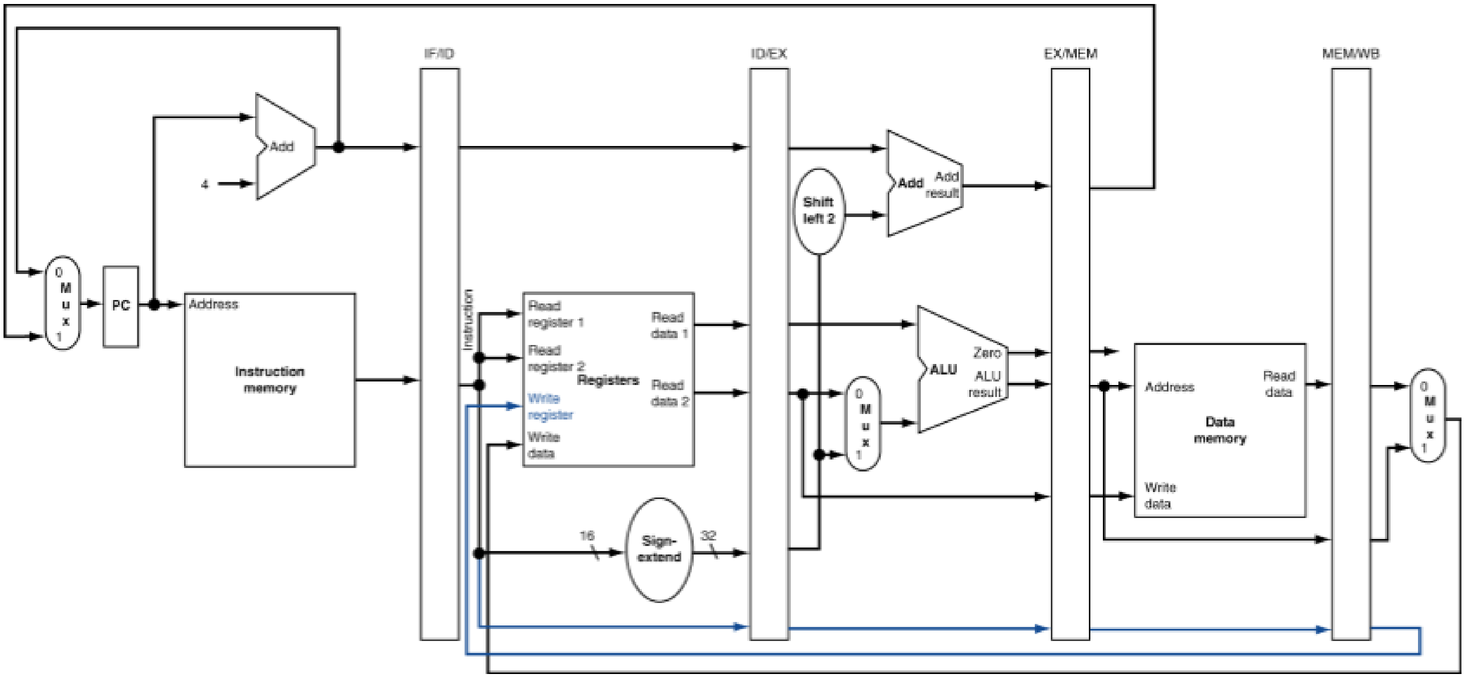
lw

Write back



Wrong register number

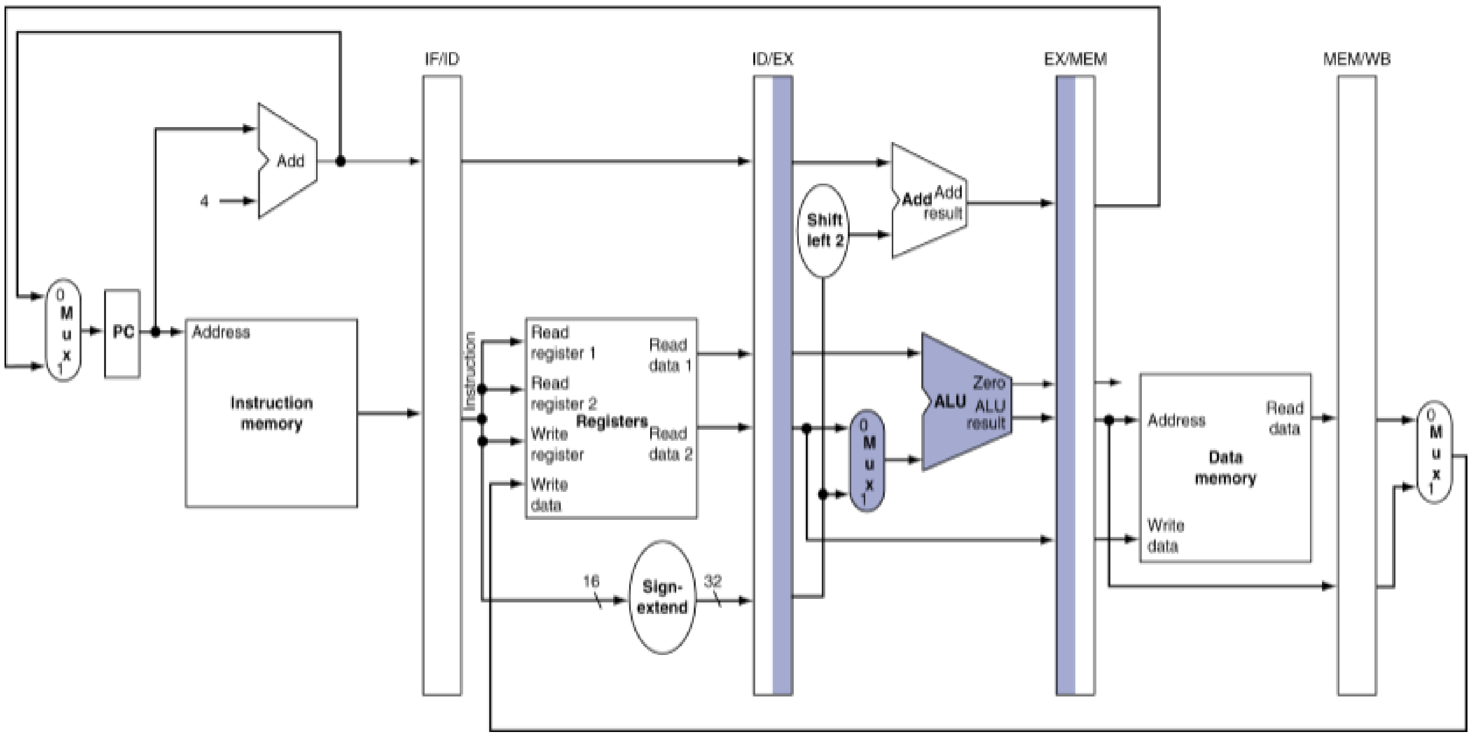
Corrected Datapath for Load



EX for Store

sw

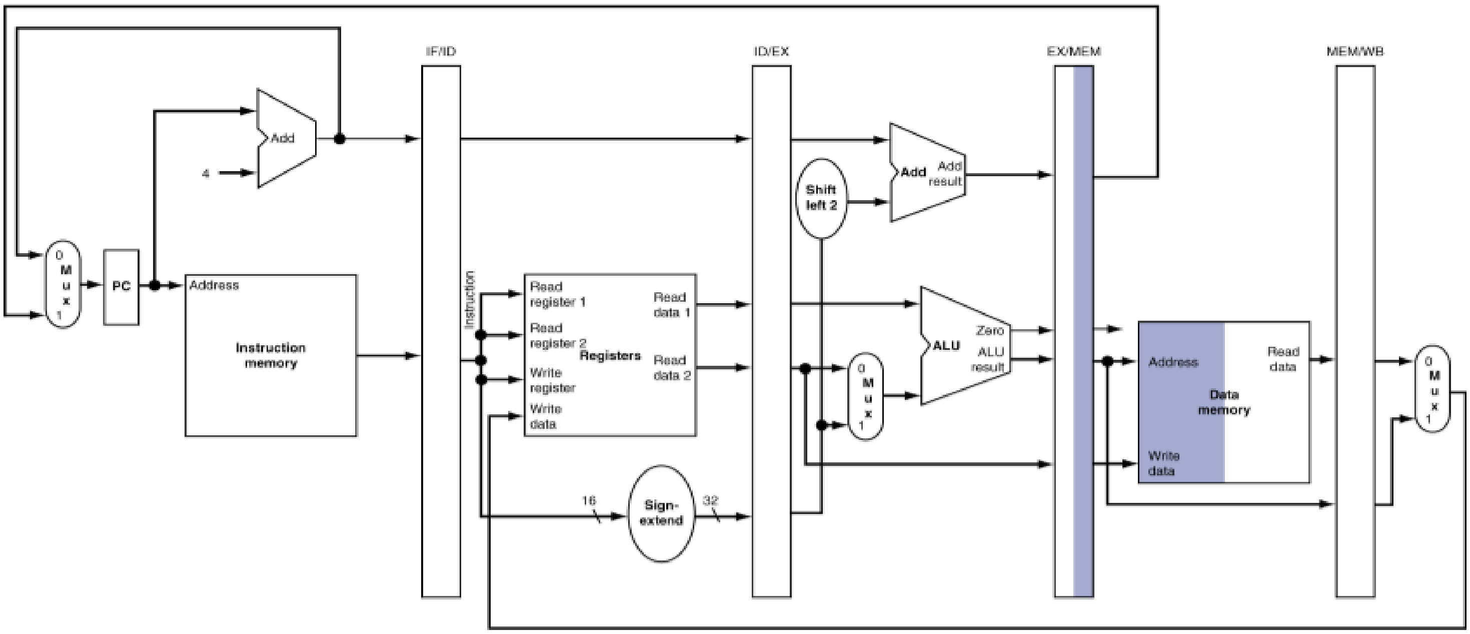
Execution



MEM for Store

sw

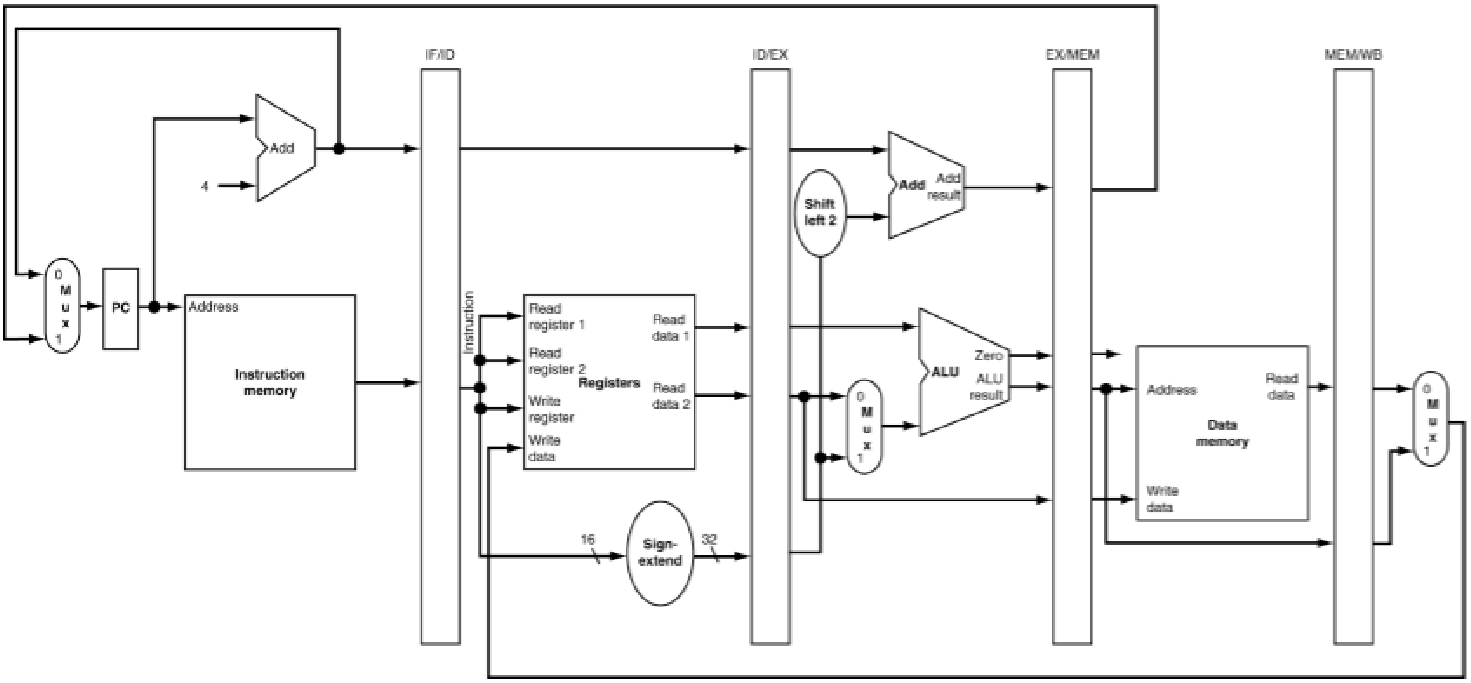
Memory



WB for Store

sw

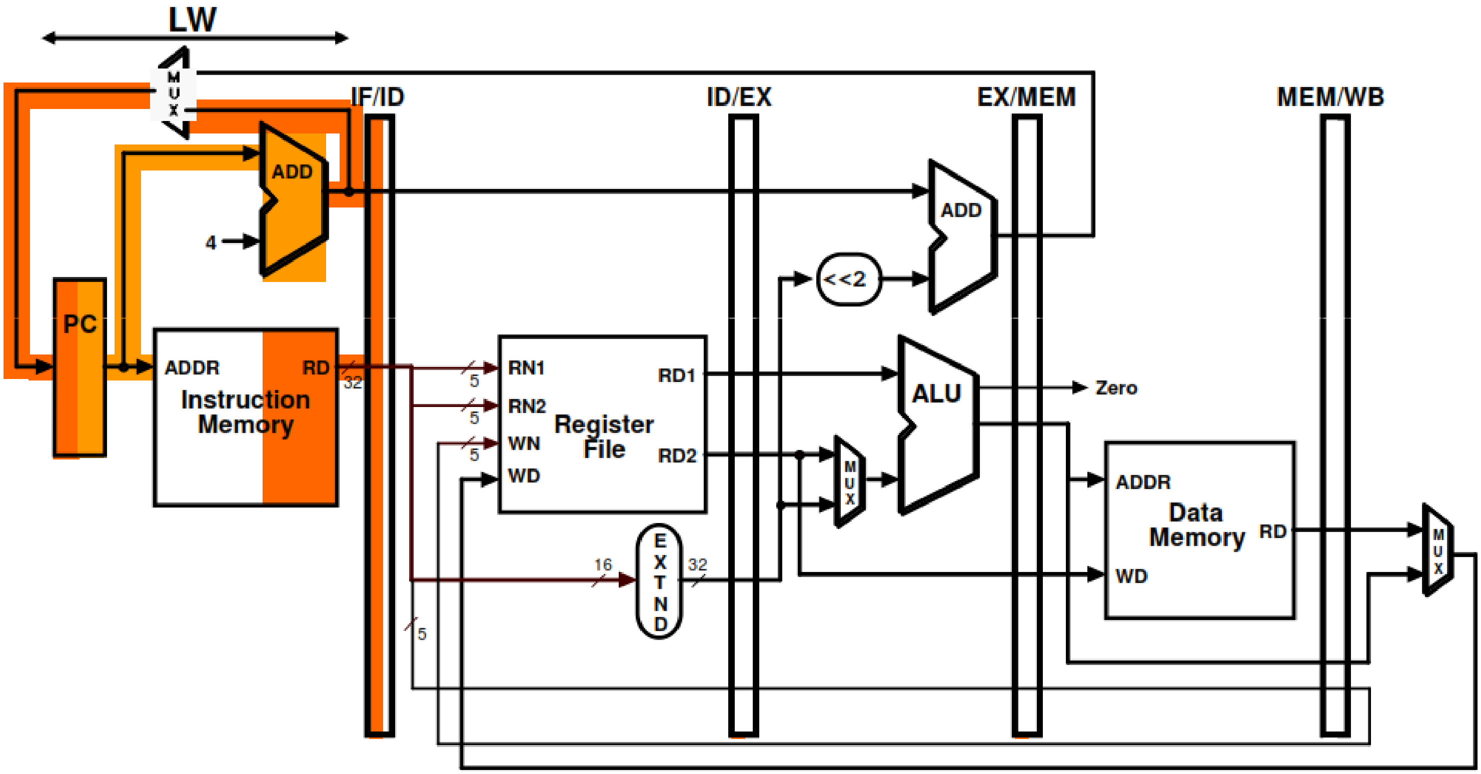
Write-back



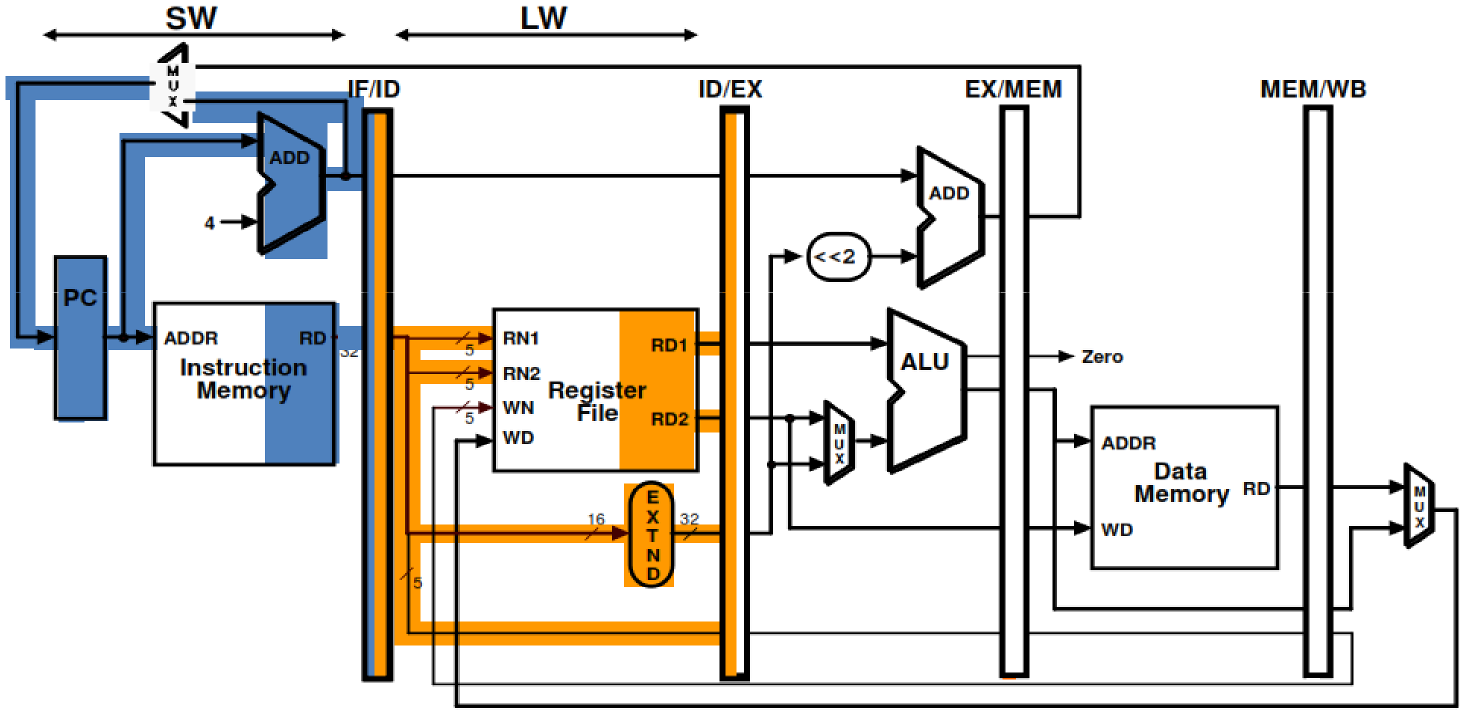
- **Example:**
- Consider the following instruction sequence:

```
lw  $t0,  10($t1)
sw  $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```

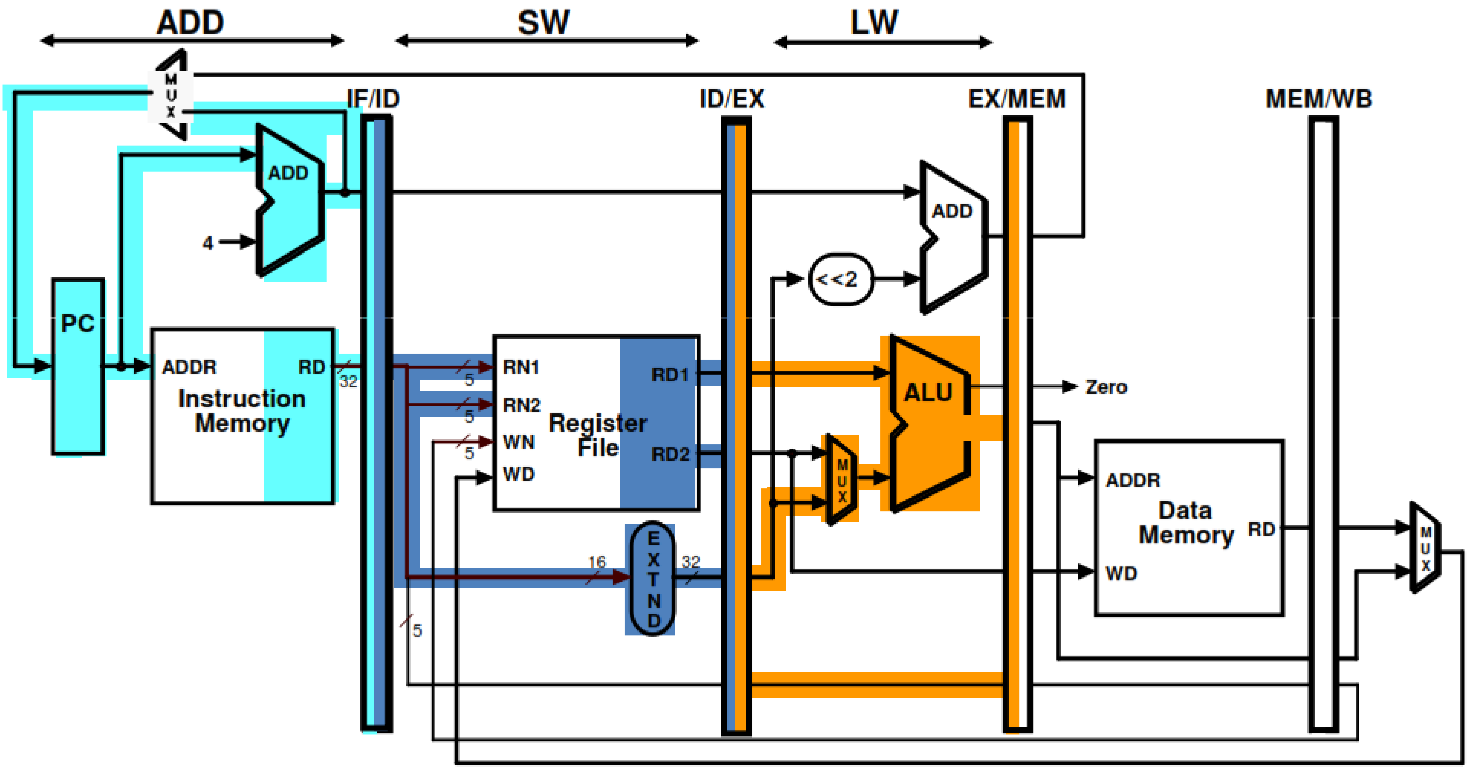
- **Clock Cycle 1:**



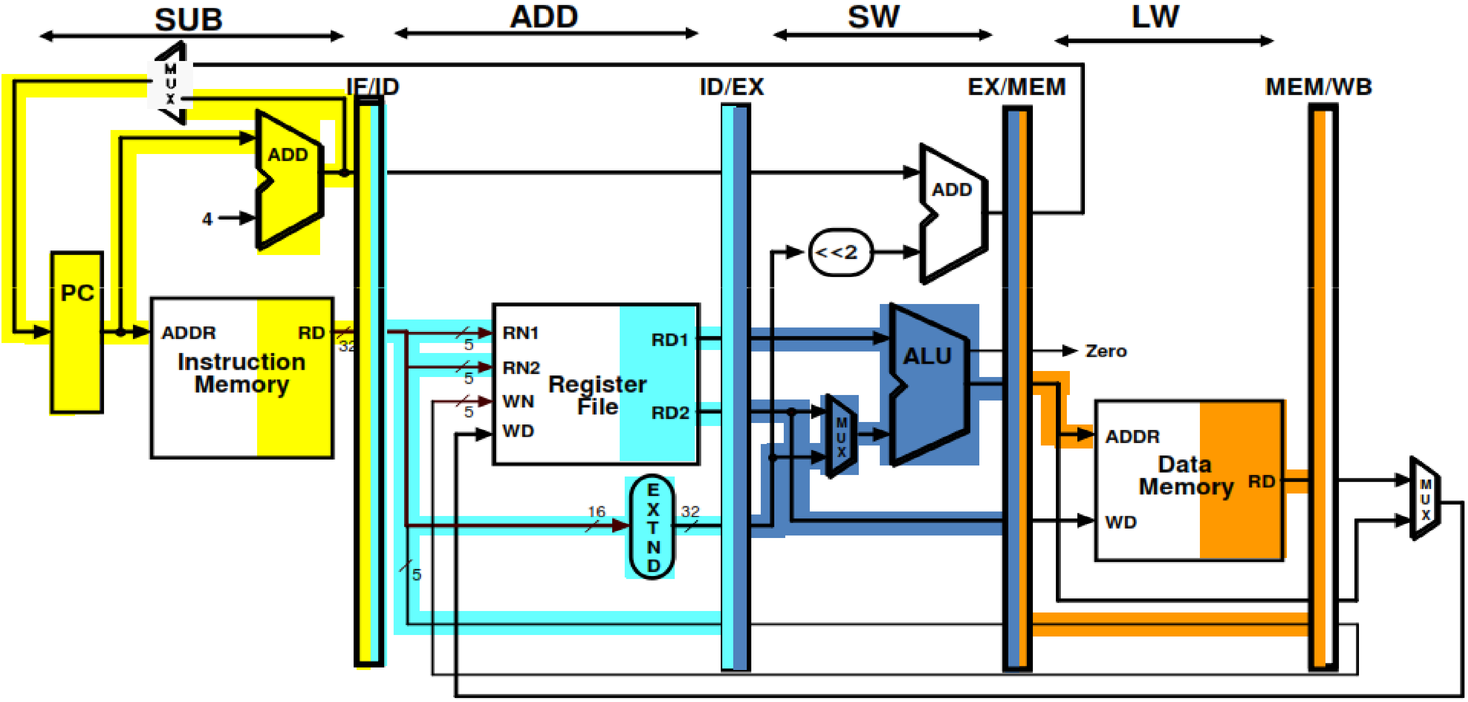
❑ Clock Cycle 2:



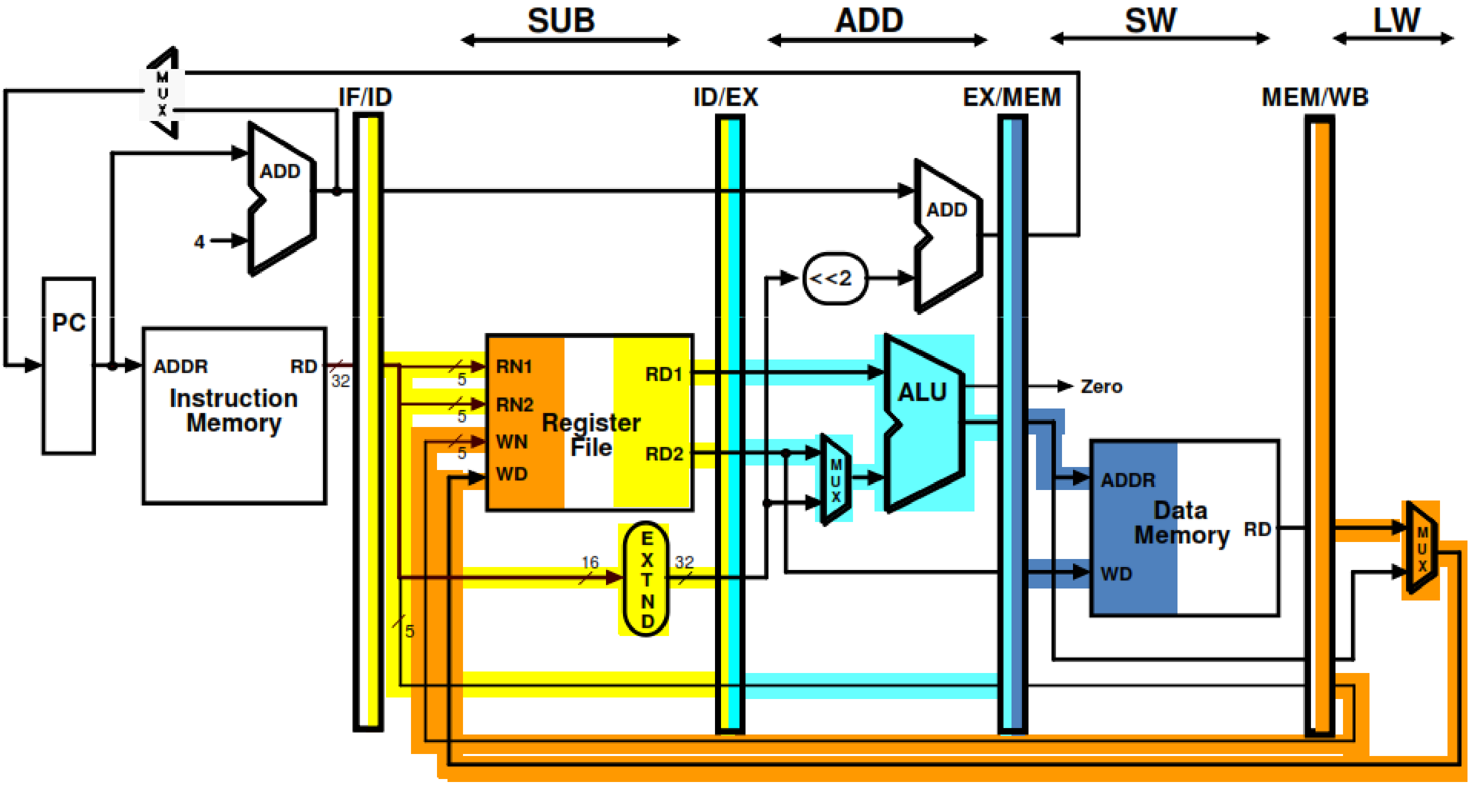
❑ Clock Cycle 3:



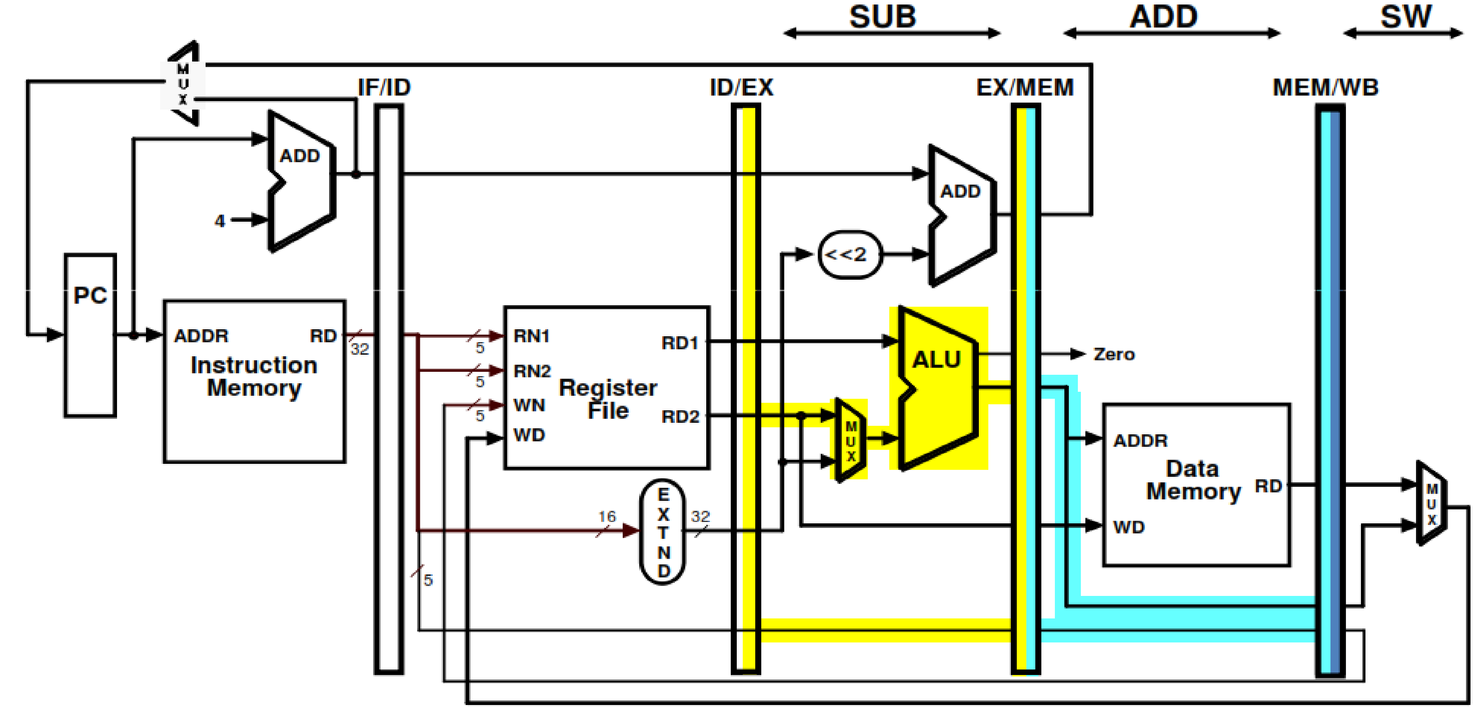
❑ Clock Cycle 4:



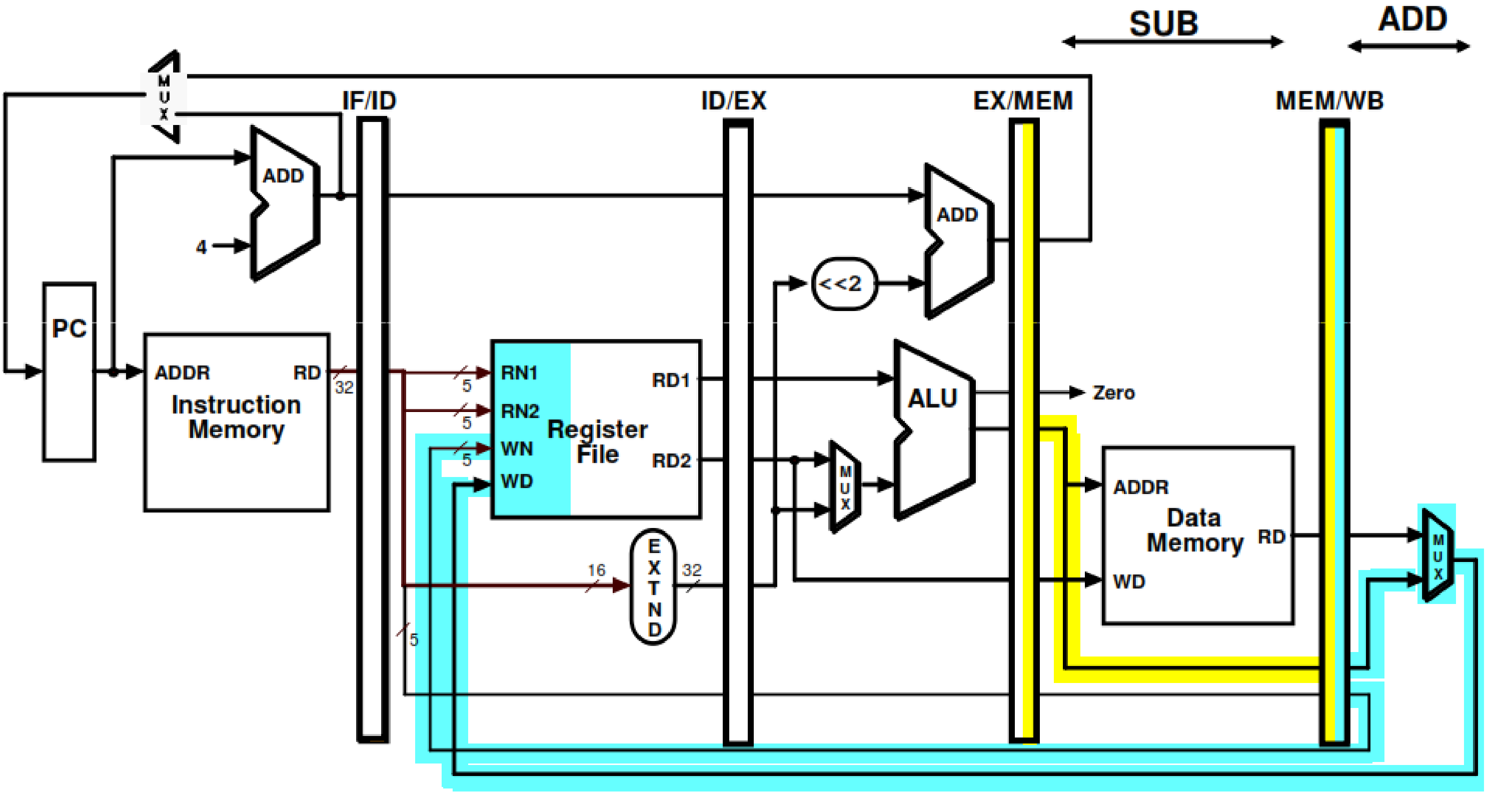
❑ Clock Cycle 5:



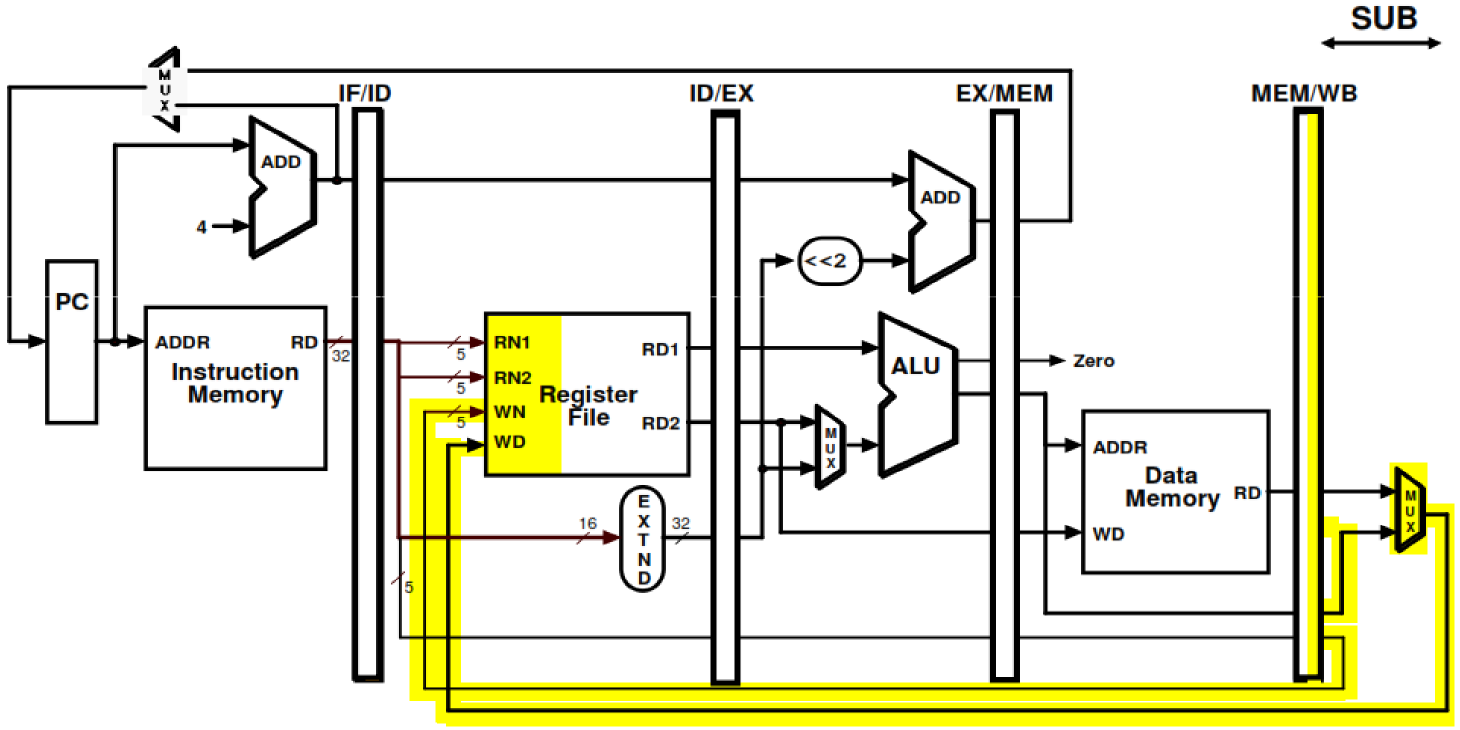
❑ Clock Cycle 6:



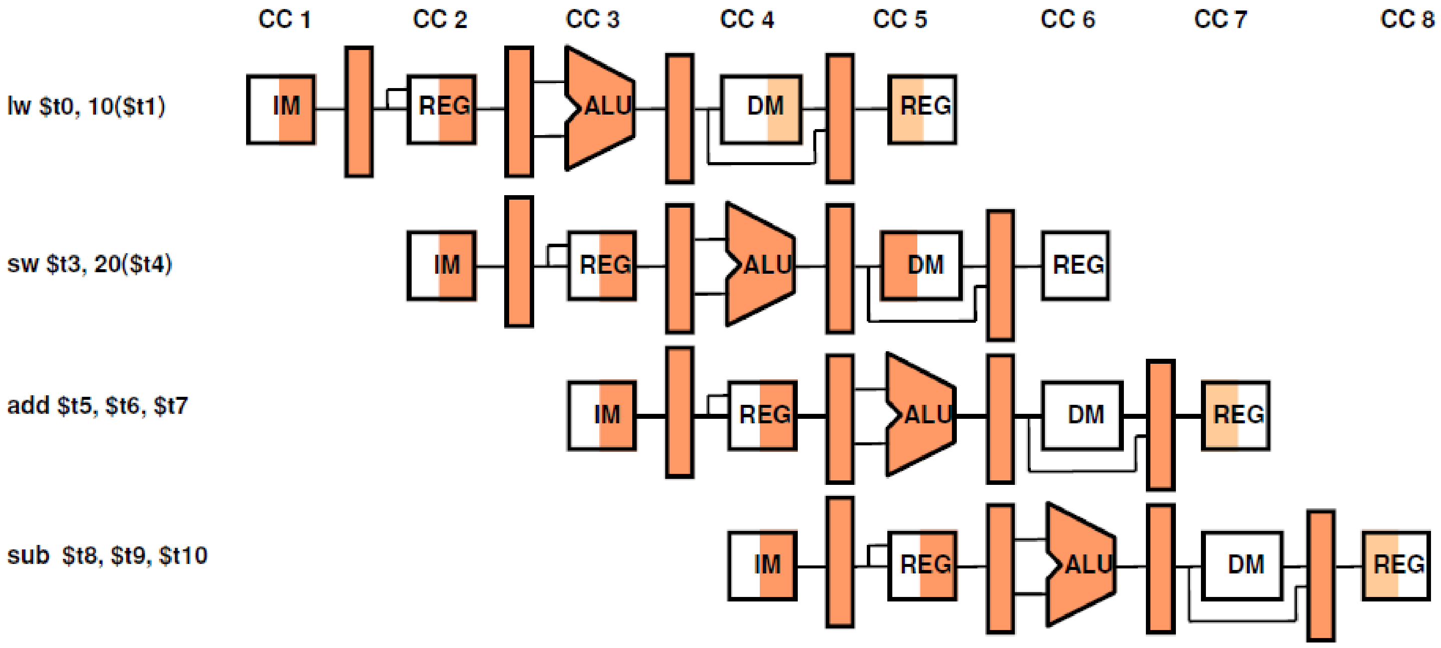
❑ Clock Cycle 7:



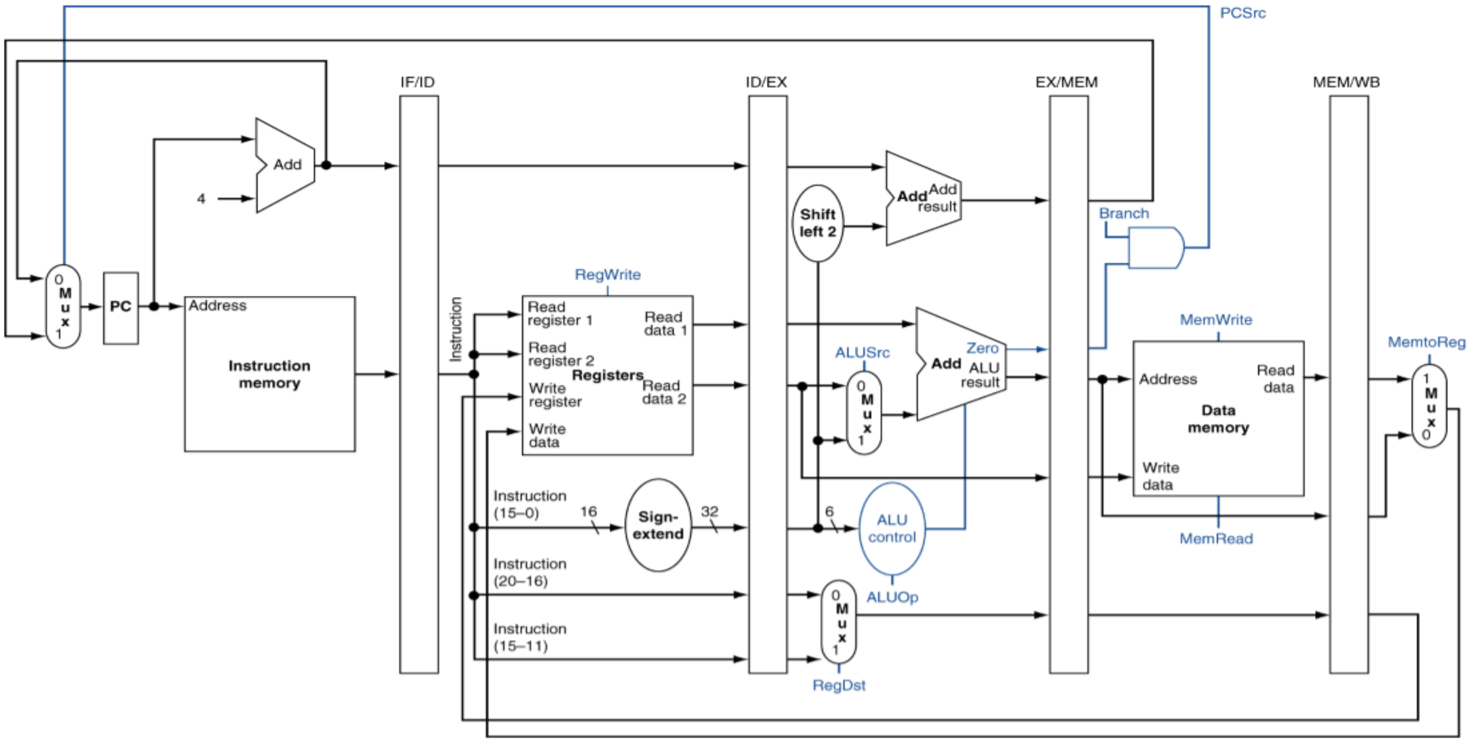
❑ Clock Cycle 8:



❑ Form showing resource usage



Pipelined Control (Simplified)



Same control signals used in the single-cycle datapath

- **As was the case for the single-cycle implementation,** we assume that the PC is written on each clock cycle, so there is no separate write signal for the PC. By the same argument, there **are no separate write signals** for the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB), since the pipeline registers are also **written during each clock cycle**.

- To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.

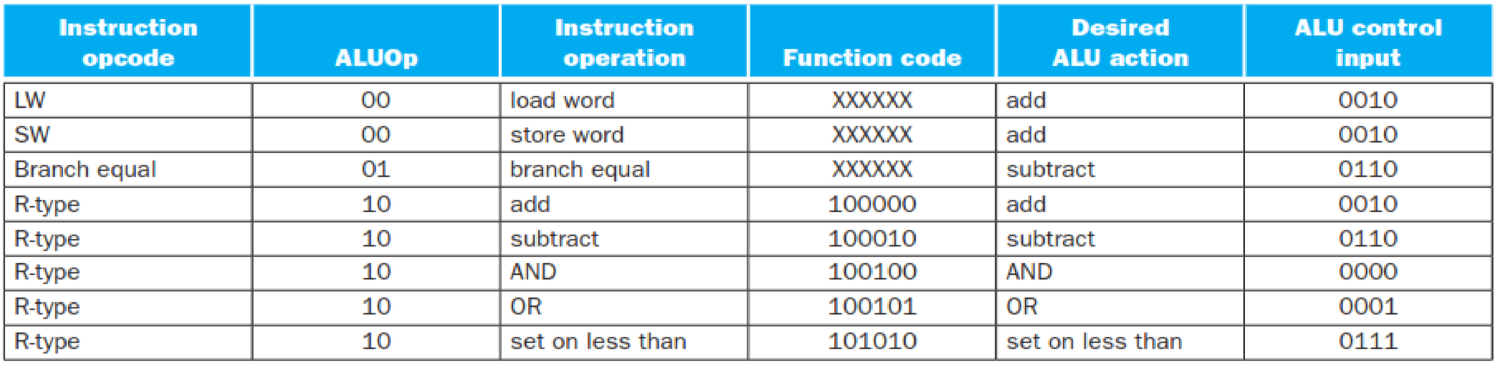
1. Instruction fetch: The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.

2. Instruction decode/register file read: As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.

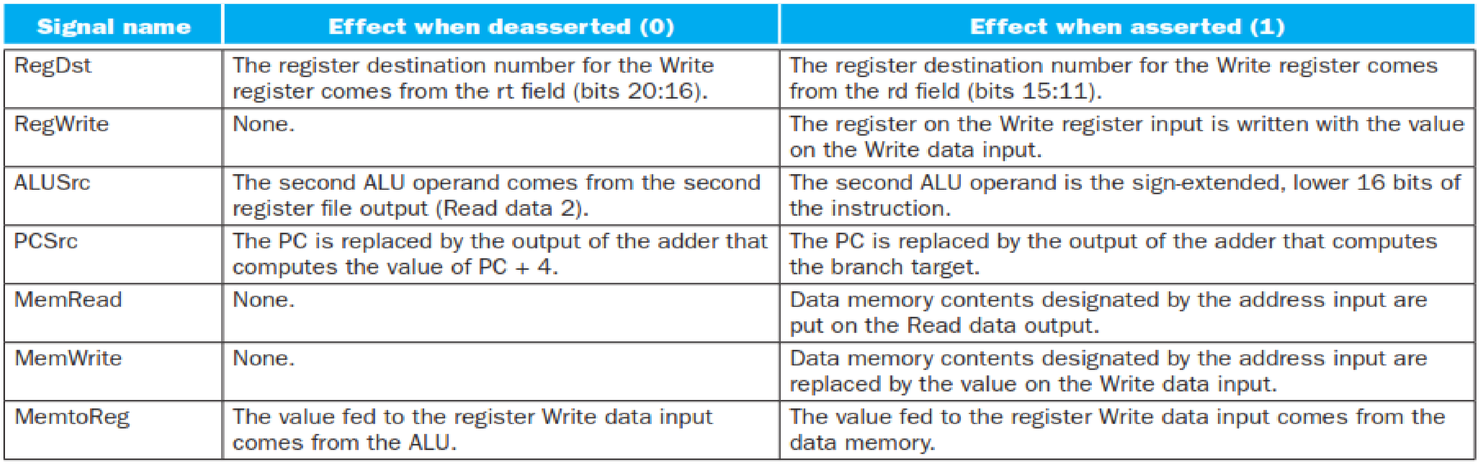
3. Execution/address calculation: The signals to be set are **RegDst**, **ALUOp**, and **ALUSrc**. The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.

4. Memory access: The control lines set in this stage are **Branch(PCSrc)**, **MemRead**, and **MemWrite**. The branch equal, load, and store instructions set these signals, respectively. Recall that PCSrc selects the next sequential address unless control asserts Branch and the ALU result was 0.

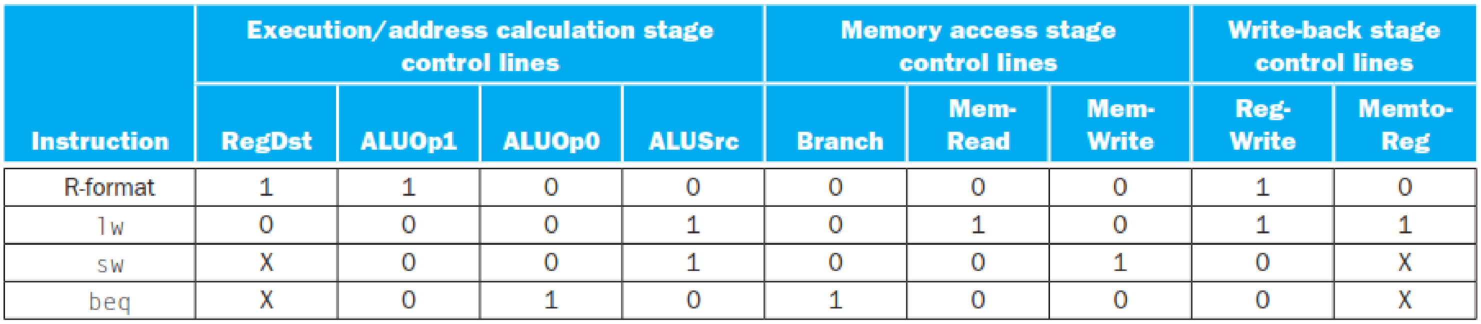
5. Write-back: The two control lines are **MemtoReg**, which decides between sending the ALU result or the memory value to the register file, and **RegWrite**, which writes the chosen value.



Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111



Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

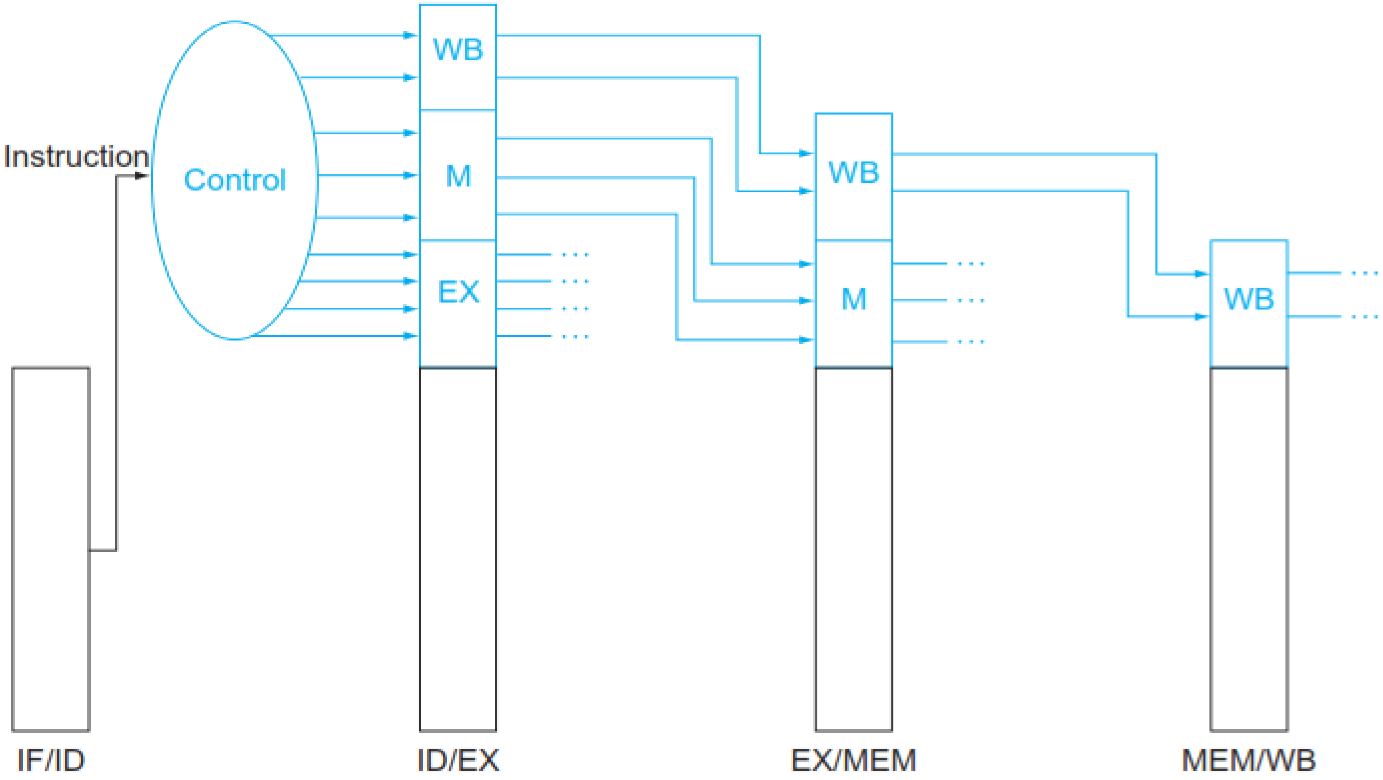


	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

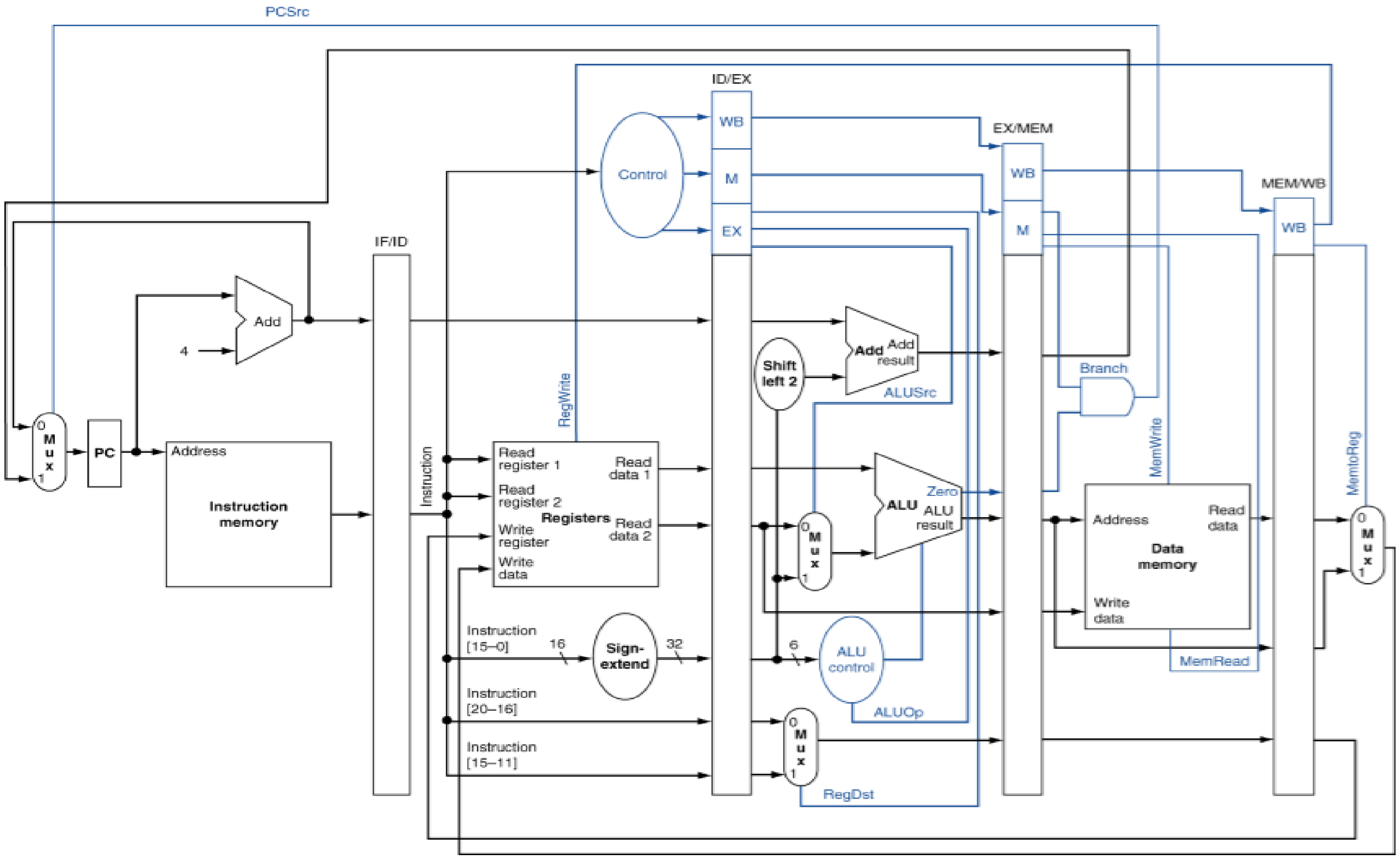
What about control signals?

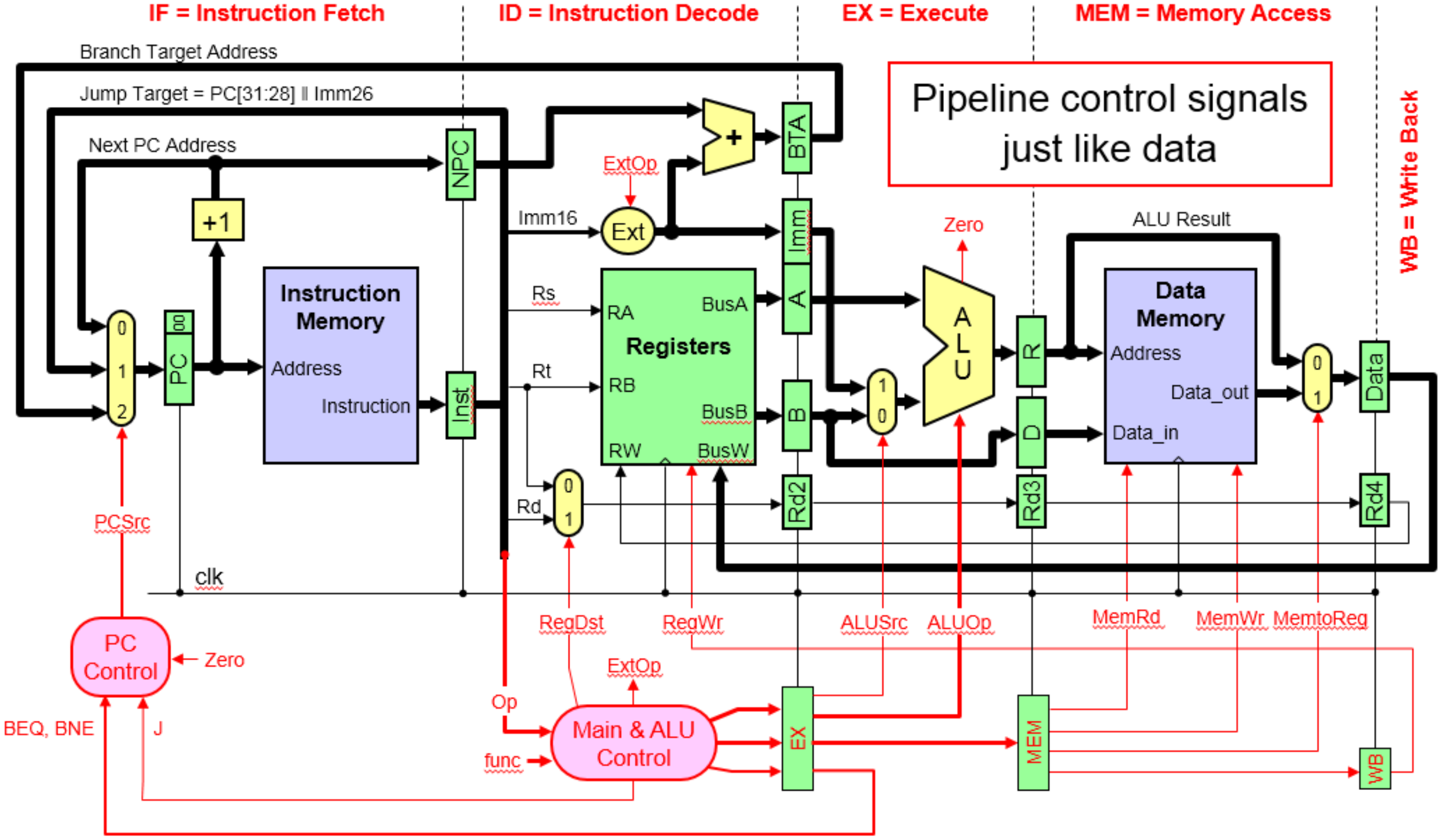
- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- But just like before, some of the control signals will not be needed until some later stage and clock cycle.
- These signals must be propagated through the pipeline until they reach the appropriate stage. We can just pass them in the pipeline registers, along with the other data.
- Control signals can be categorized by the pipeline stage that uses them.

Stage	Control signals needed		
EX	ALUSrc	ALUOp	RegDst
MEM	MemRead	MemWrite	PCSrc
WB	RegWrite	MemToReg	



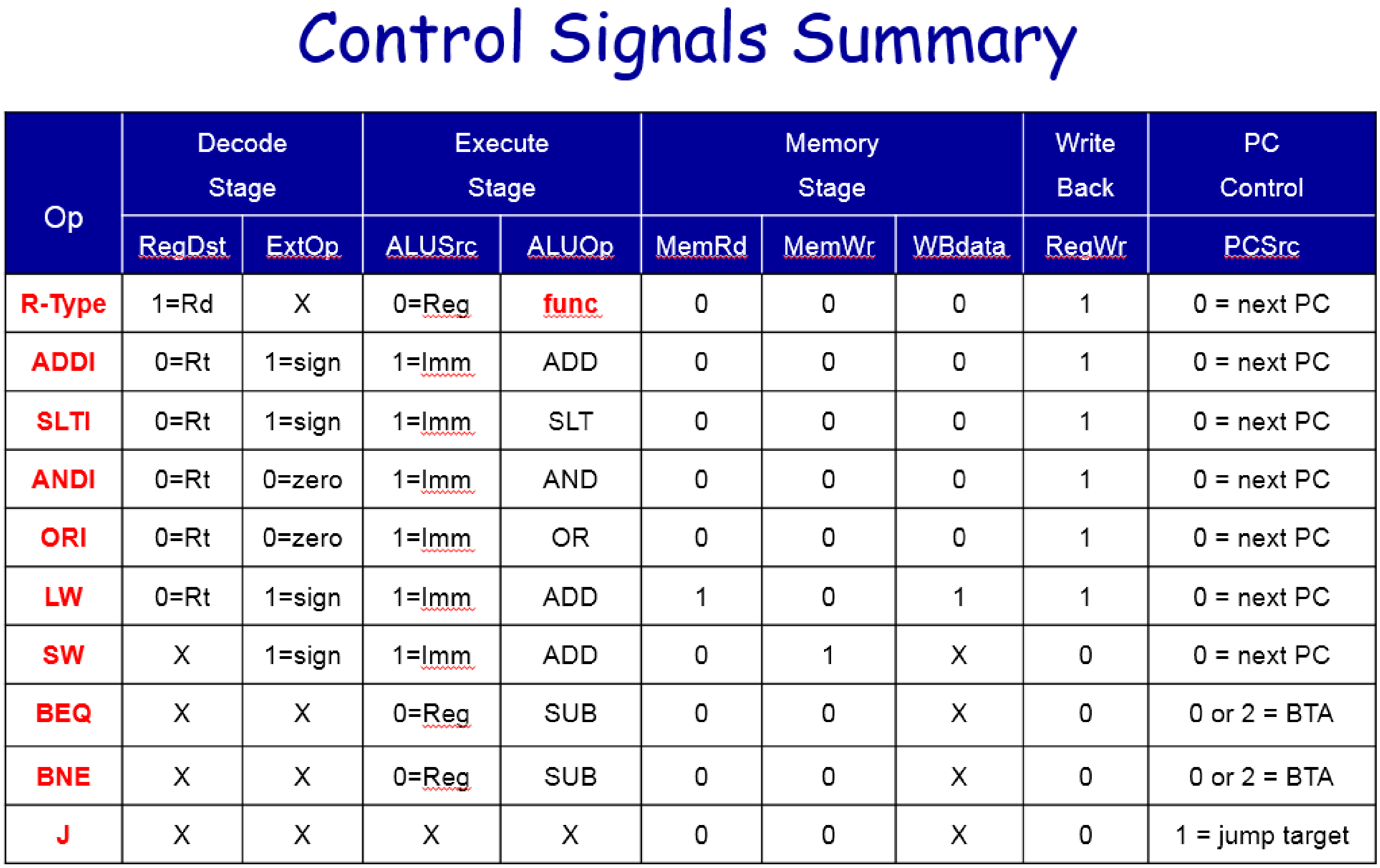
Pipelined Datapath with Control





Pipeline control signals just like data

- ❖ ID stage generates all the control signals
- ❖ Pipeline the control signals as the instruction moves
  - ✧ Extend the pipeline registers to include the control signals
- ❖ Each stage uses some of the control signals
  - ✧ Instruction Decode and Register Read
    - ▪ Control signals are generated
    - ▪ RegDst and ExtOp are used in this stage, J (Jump) is used by PC control
  - ✧ Execution Stage => ALUSrc, ALUOp, BEQ, BNE
    - ▪ ALU generates zero signal for PC control logic (Branch Control)
  - ✧ Memory Stage => MemRd and MemWr
  - ✧ Write Back Stage => RegWr and MemtoReg

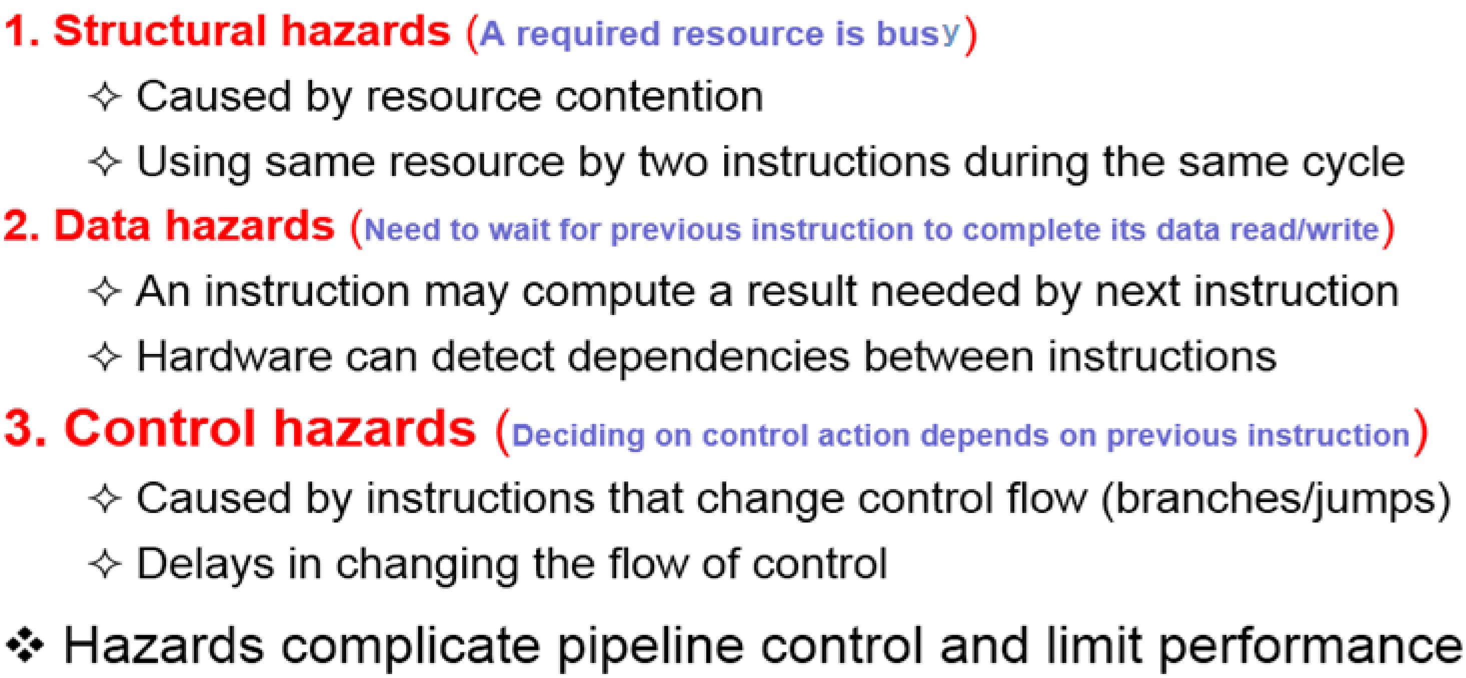


Control Signals Summary

Op	Decode Stage		Execute Stage		Memory Stage			Write Back	PC Control
	RegDst	ExtOp	ALUSrc	ALUOp	MemRd	MemWr	WBdata	RegWr	PCSrc
R-Type	1=Rd	X	0=Reg	func	0	0	0	1	0 = next PC
ADDI	0=Rt	1=sign	1=Imm	ADD	0	0	0	1	0 = next PC
SLTI	0=Rt	1=sign	1=Imm	SLT	0	0	0	1	0 = next PC
ANDI	0=Rt	0=zero	1=Imm	AND	0	0	0	1	0 = next PC
ORI	0=Rt	0=zero	1=Imm	OR	0	0	0	1	0 = next PC
LW	0=Rt	1=sign	1=Imm	ADD	1	0	1	1	0 = next PC
SW	X	1=sign	1=Imm	ADD	0	1	X	0	0 = next PC
BEQ	X	X	0=Reg	SUB	0	0	X	0	0 or 2 = BTA
BNE	X	X	0=Reg	SUB	0	0	X	0	0 or 2 = BTA
J	X	X	X	X	0	0	X	0	1 = jump target

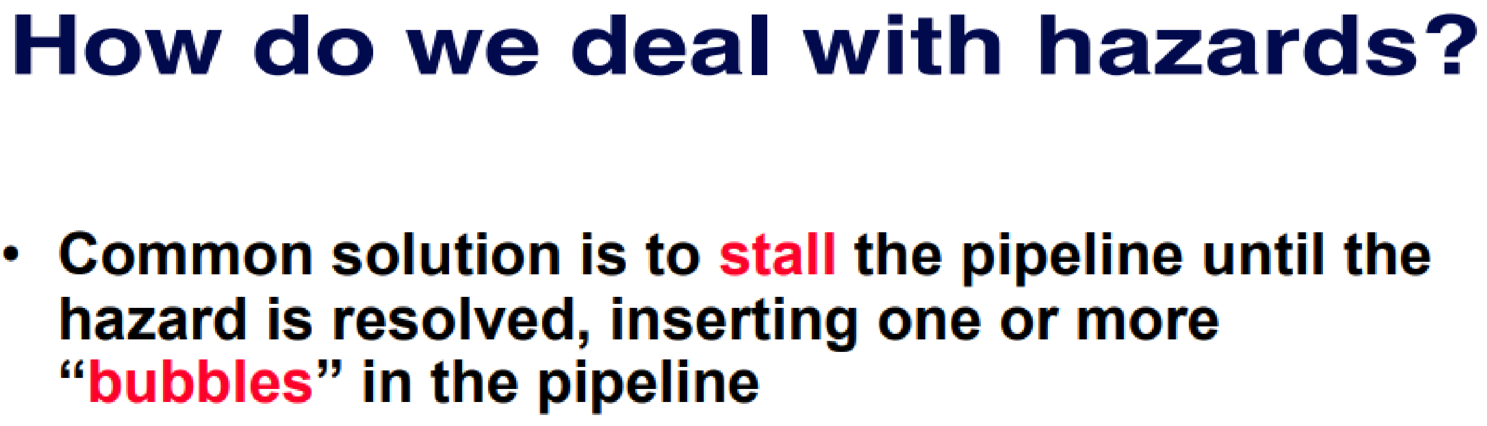
Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three different types.



1. **Structural hazards** (A required resource is busy)
   - ✧ Caused by resource contention
   - ✧ Using same resource by two instructions during the same cycle
2. **Data hazards** (Need to wait for previous instruction to complete its data read/write)
   - ✧ An instruction may compute a result needed by next instruction
   - ✧ Hardware can detect dependencies between instructions
3. **Control hazards** (Deciding on control action depends on previous instruction)
   - ✧ Caused by instructions that change control flow (branches/jumps)
   - ✧ Delays in changing the flow of control

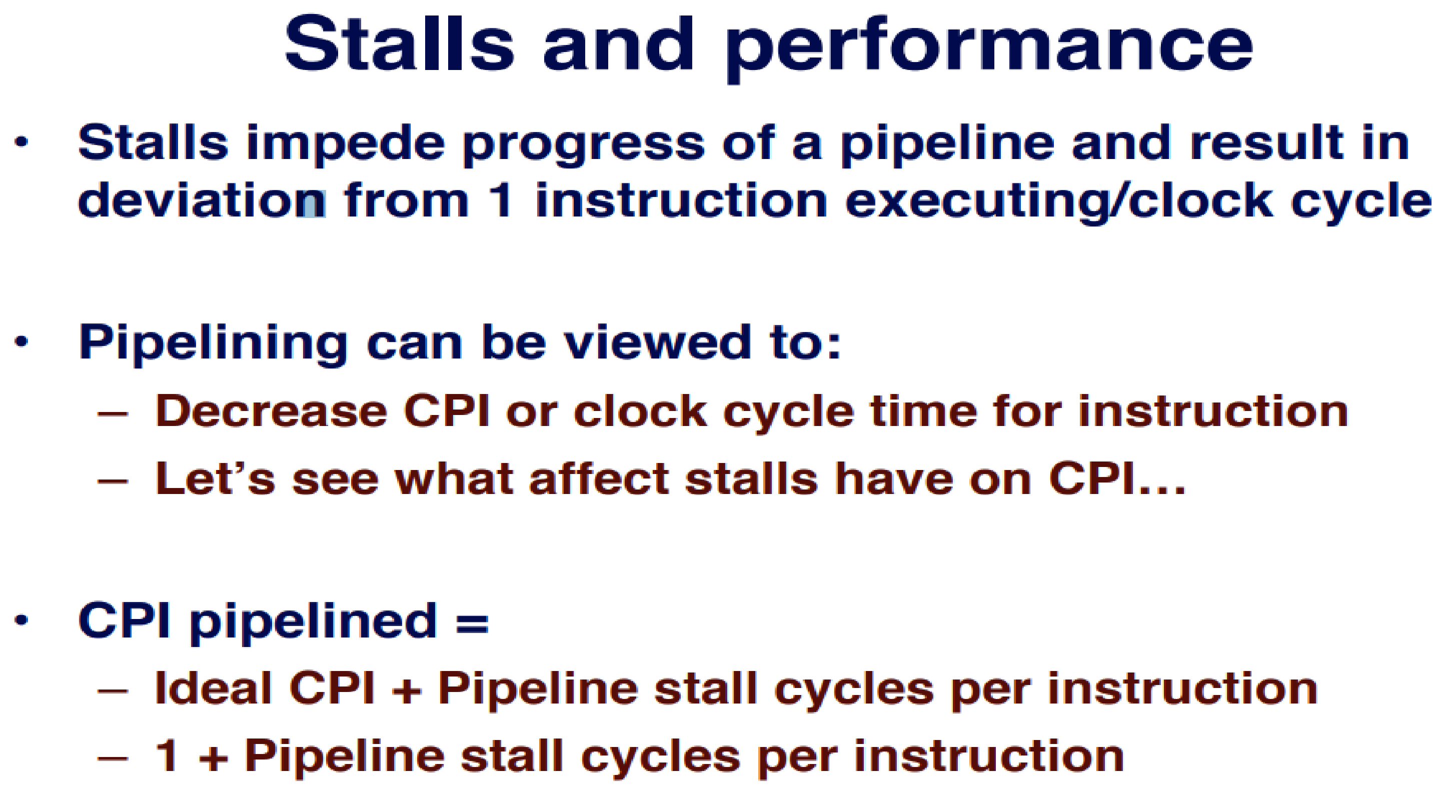
❖ Hazards complicate pipeline control and limit performance



How do we deal with hazards?

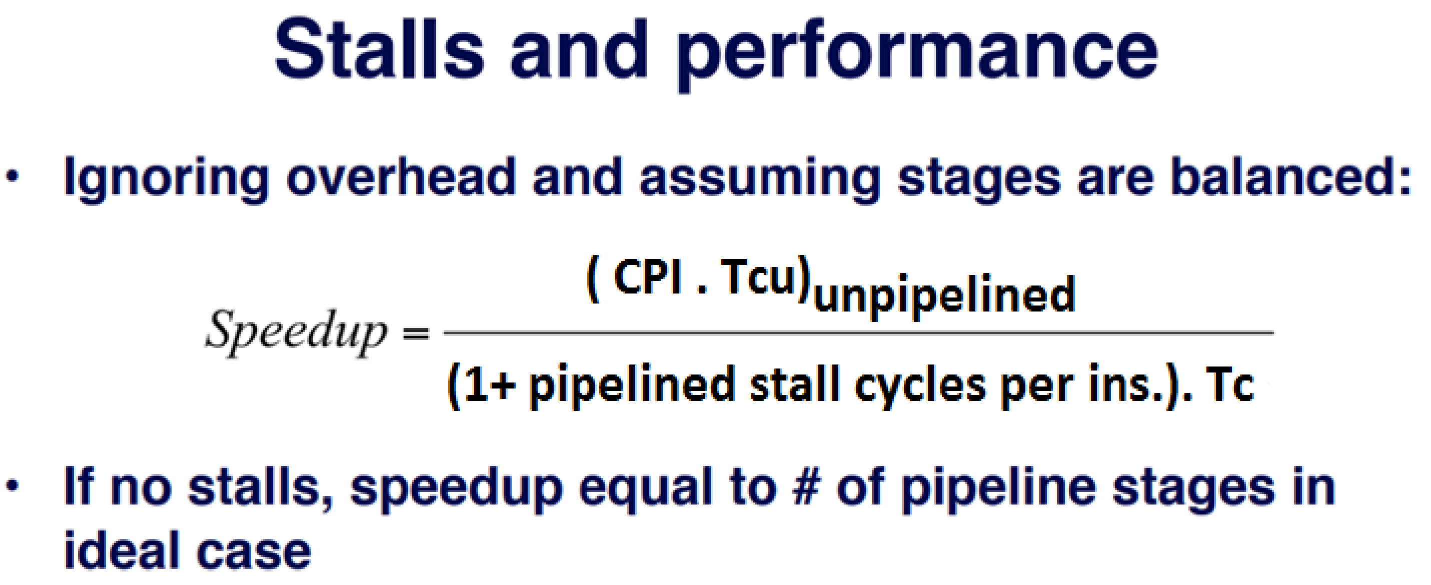
- **Common solution is to stall the pipeline until the hazard is resolved, inserting one or more "bubbles" in the pipeline**

- **In the design of pipelined computer processors, a pipeline stall is a delay in execution of an instruction in order to resolve a hazard. Such an event is often called a bubble, by analogy with an air bubble in a fluid pipe.**



Stalls and performance

- Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle
- Pipelining can be viewed to:
  - Decrease CPI or clock cycle time for instruction
  - Let's see what affect stalls have on CPI…
- CPI pipelined =
  - Ideal CPI + Pipeline stall cycles per instruction
  - 1 + Pipeline stall cycles per instruction



Stalls and performance

- Ignoring overhead and assuming stages are balanced:

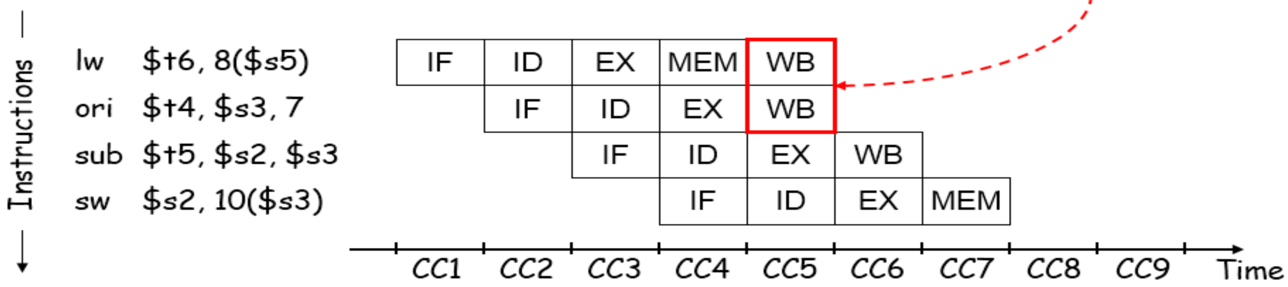
$$Speedup = \frac{(\text{ CPI . Tcu})_{\text{unpipelined}}}{(1+\text{ pipelined stall cycles per ins.}).\ \text{Tc}}$$

- If no stalls, speedup equal to # of pipeline stages in ideal case

Structural Hazards

- **Problem**
  - Attempt to use the same hardware resource by two different instructions during the same clock cycle
- **Example**
  - Writing back ALU result in stage 4
  - Conflict with writing load data in stage 5

Structural Hazard
Two instructions are attempting to write the register file during same cycle

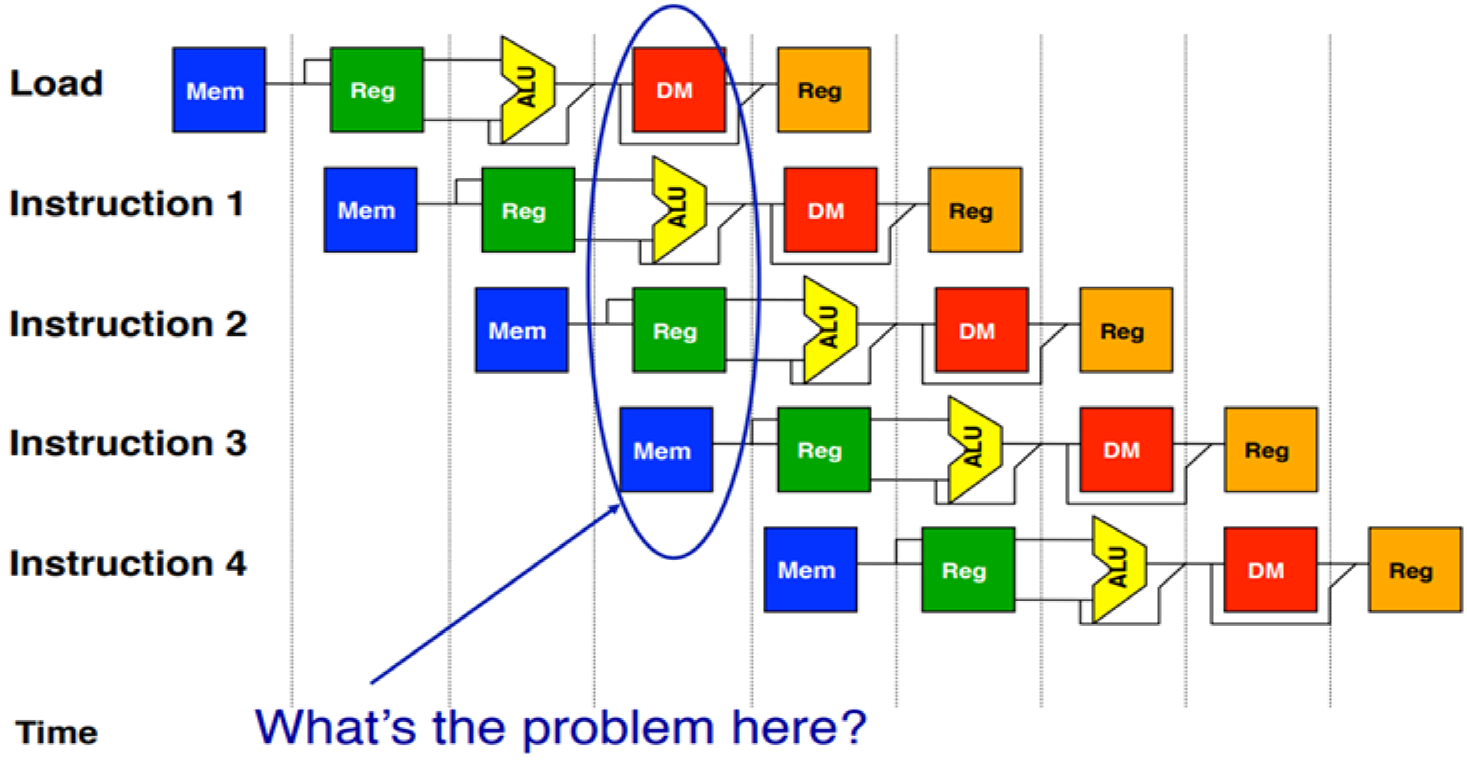


Instructions	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
lw $t6, 8($s5)	IF	ID	EX	MEM	WB				
ori $t4, $s3, 7		IF	ID	EX	WB				
sub $t5, $s2, $s3			IF	ID	EX	WB			
sw $s2, 10($s3)				IF	ID	EX	MEM		

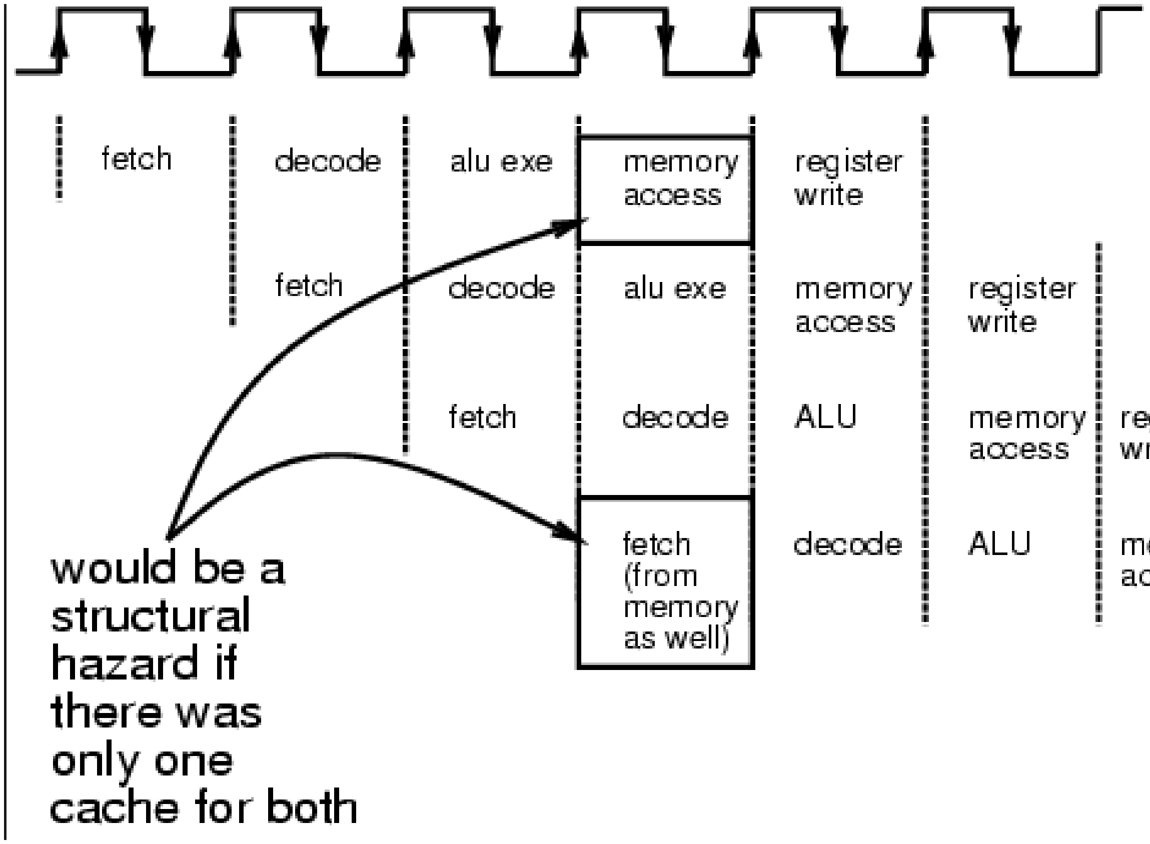
Resolving Structural Hazards

- **Serious Hazard:**
  - Hazard cannot be ignored
- **Solution 1: Delay Access to Resource**
  - Must have mechanism to delay instruction access to resource
  - Delay all write backs to the register file to stage 5
    - ALU instructions bypass stage 4 (memory) without doing anything
- **Solution 2: Add more hardware resources (more costly)**
  - Add more hardware to eliminate the structural hazard
  - Redesign the register file to have two write ports
    - First write port can be used to write back ALU results in stage 4
    - Second write port can be used to write back load data in stage 5

One Memory Port/Structural Hazards



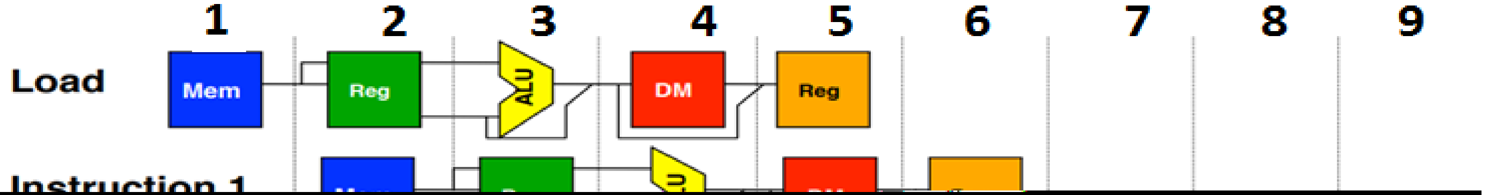
What’s the problem here?



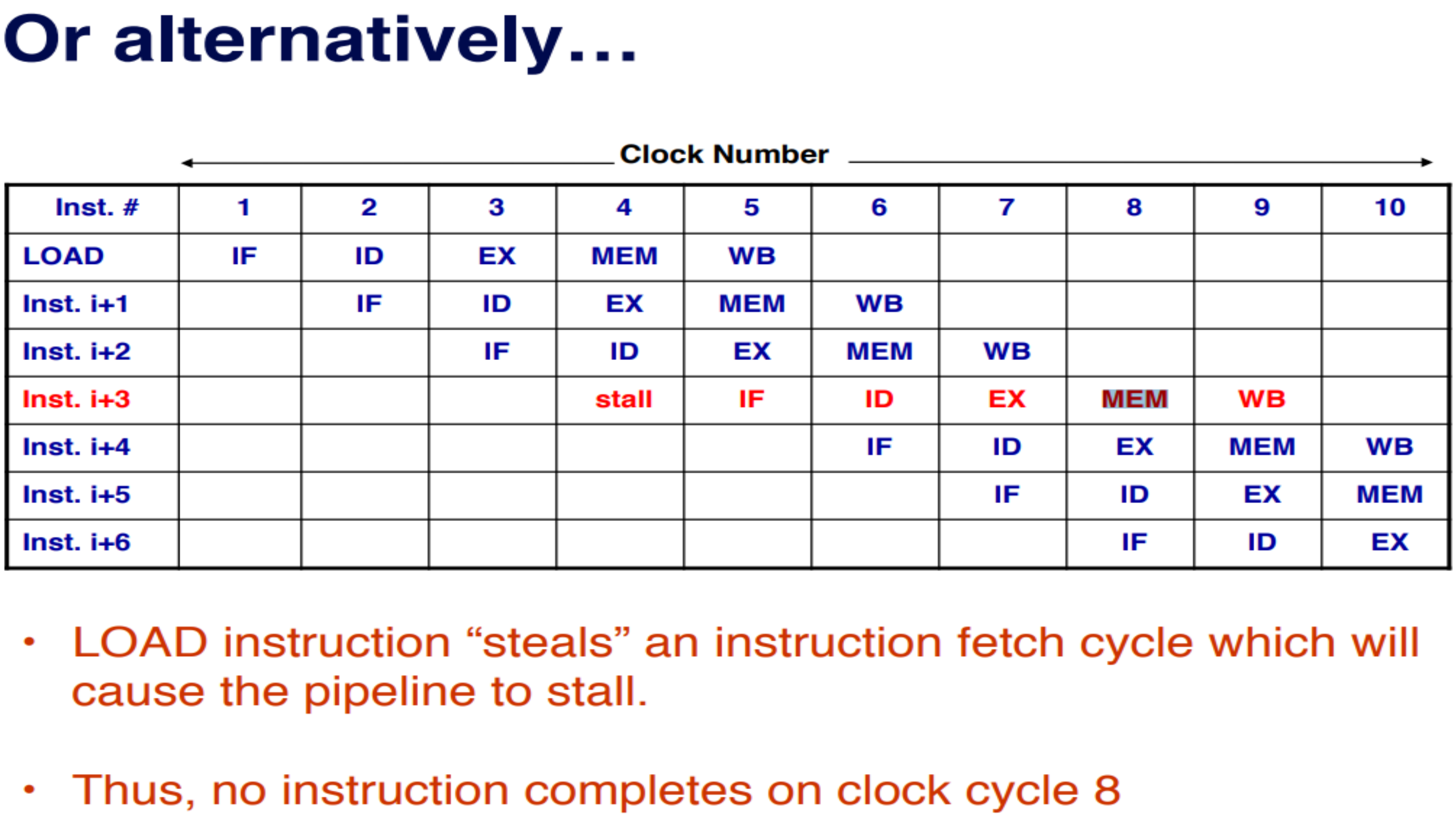
fetch	decode	alu exe	memory access	register write		
	fetch	decode	alu exe	memory access	register write	
		fetch	decode	ALU	memory access	register write
			fetch (from memory as well)	decode	ALU	memory access

would be a structural hazard if there was only one cache for both

How is it resolved?



[redacted]



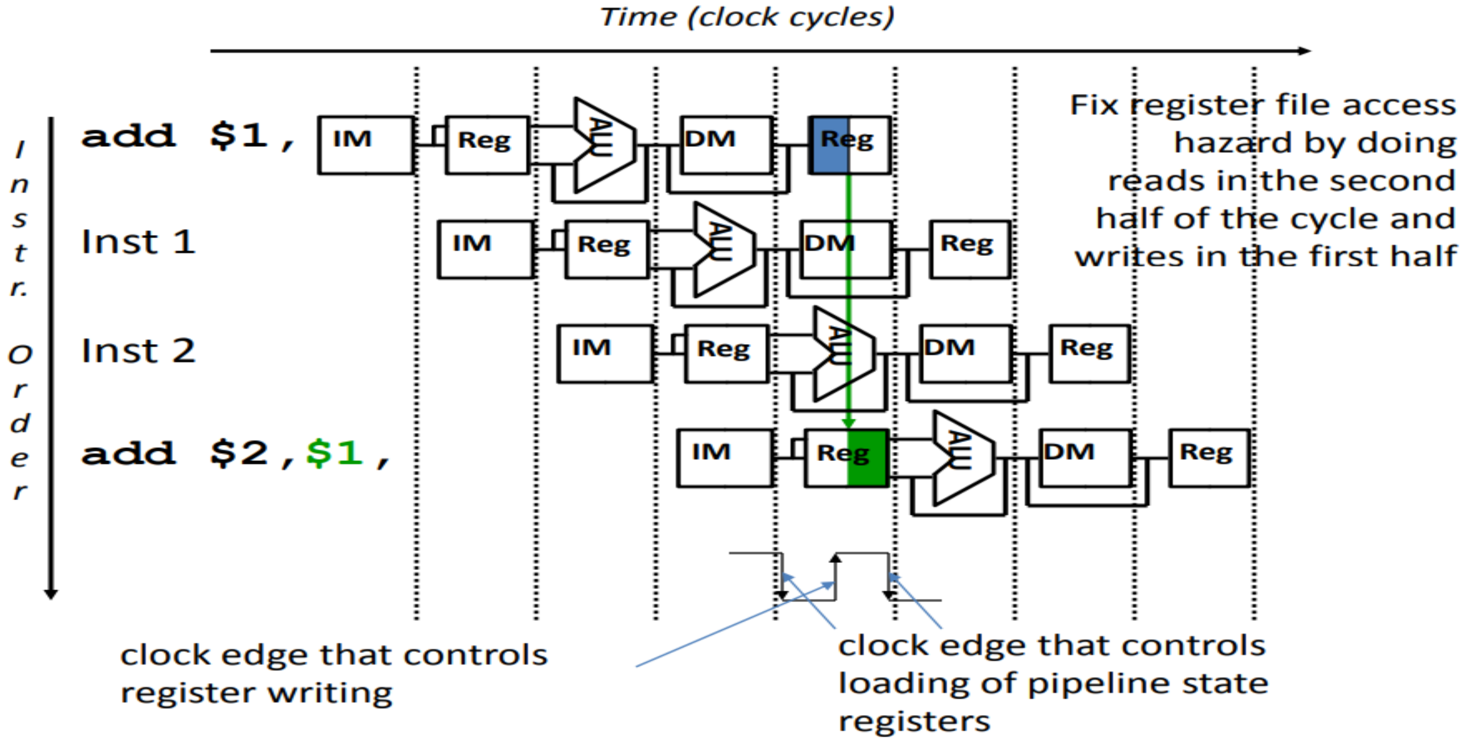
Or alternatively…

Clock Number

Inst. #	1	2	3	4	5	6	7	8	9	10
LOAD	IF	ID	EX	MEM	WB					
Inst. i+1		IF	ID	EX	MEM	WB				
Inst. i+2			IF	ID	EX	MEM	WB			
Inst. i+3				stall	IF	ID	EX	MEM	WB	
Inst. i+4						IF	ID	EX	MEM	WB
Inst. i+5							IF	ID	EX	MEM
Inst. i+6								IF	ID	EX

- LOAD instruction “steals” an instruction fetch cycle which will cause the pipeline to stall.
- Thus, no instruction completes on clock cycle 8

How About Register File Access?



Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half

clock edge that controls register writing

clock edge that controls loading of pipeline state registers

Data Hazards

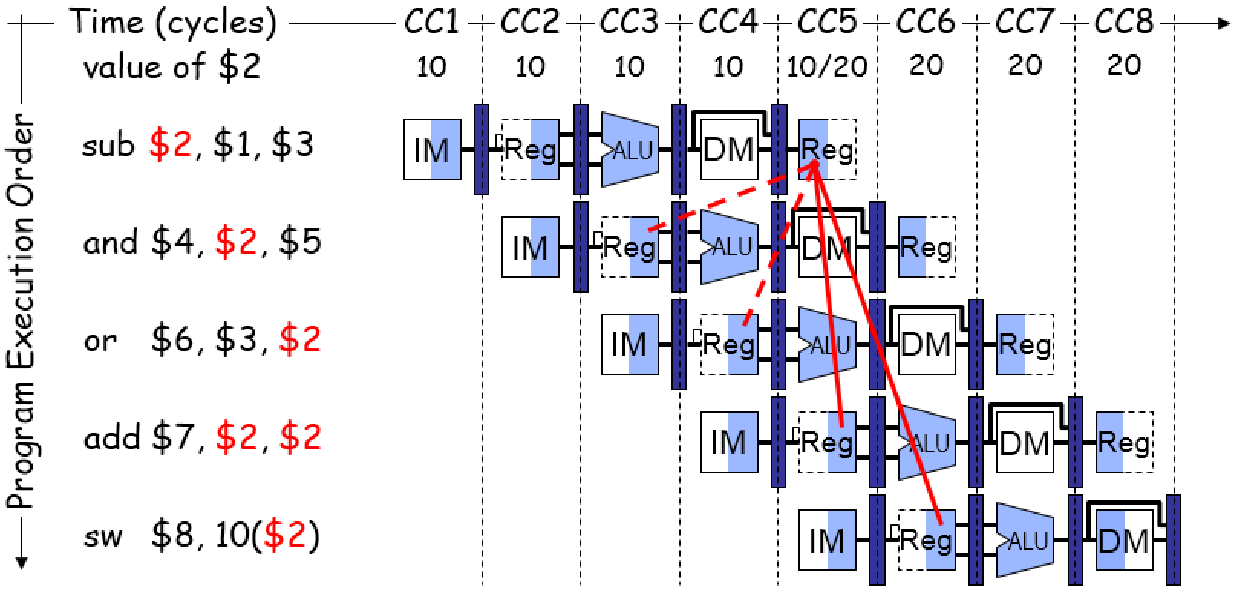
data hazard Also called a **pipeline data hazard**. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

- Dependency between instructions causes a data hazard
- The dependent instructions are close to each other
  - Pipelined execution might change the order of operand access
- Read After Write – RAW Hazard
  - Given two instructions *x* and *y*, where *x* comes before *y* …
  - Instruction *y* should read an operand after it is written by *x*
  - Called a data dependence in compiler terminology

```
x: add $1, $2, $3     # r1 is written (Fifth stage)
y: sub $4, $1, $3     # r1 is read (Second stage)
```

  - Hazard occurs when *y* reads the operand before *x* writes it

Example of a RAW Data Hazard



- ❖ Result of sub is needed by and, or, add, & sw instructions
- ❖ Instructions and & or will read old value of $2 from reg file
- ❖ During CC5, $2 is written and read – new value is read

The SUB does not write to register $2 until clock cycle 5 causeing 2 data hazards in our pipelined datapath

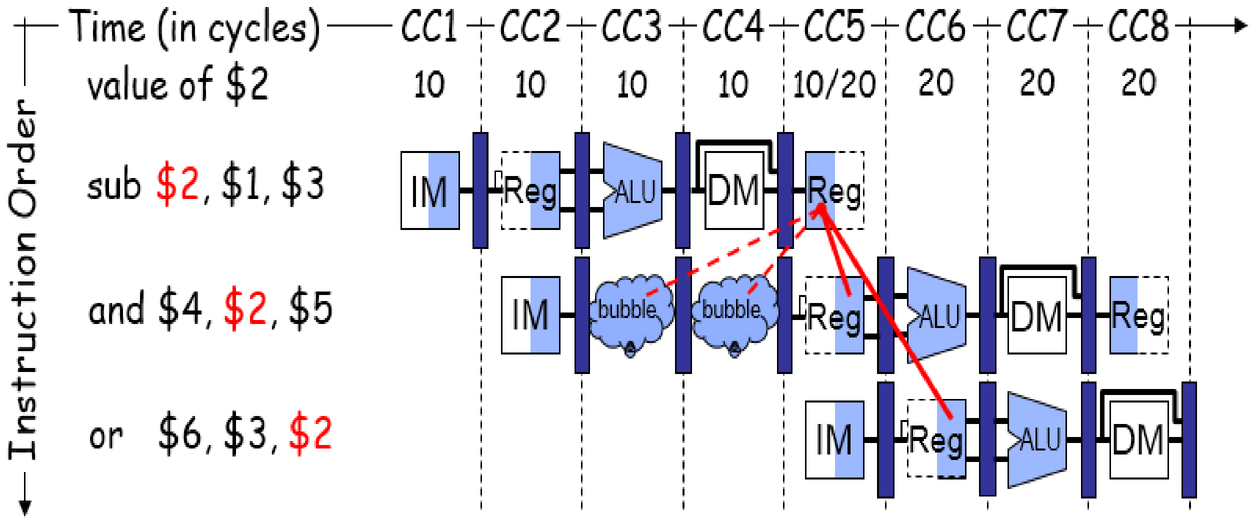
- ❖ The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this is the *old* value of $2
- ❖ Similarly, the OR instruction uses register $2 in cycle 4, again before it's actually updated by SUB

The ADD is okay, because of the register file design

- ❖ Registers are written at the beginning of a clock cycle
- ❖ The new value will be available by the end of that cycle

The SW is no problem at all, since it reads $2 after the SUB finishes

Solution 1: Stalling the Pipeline



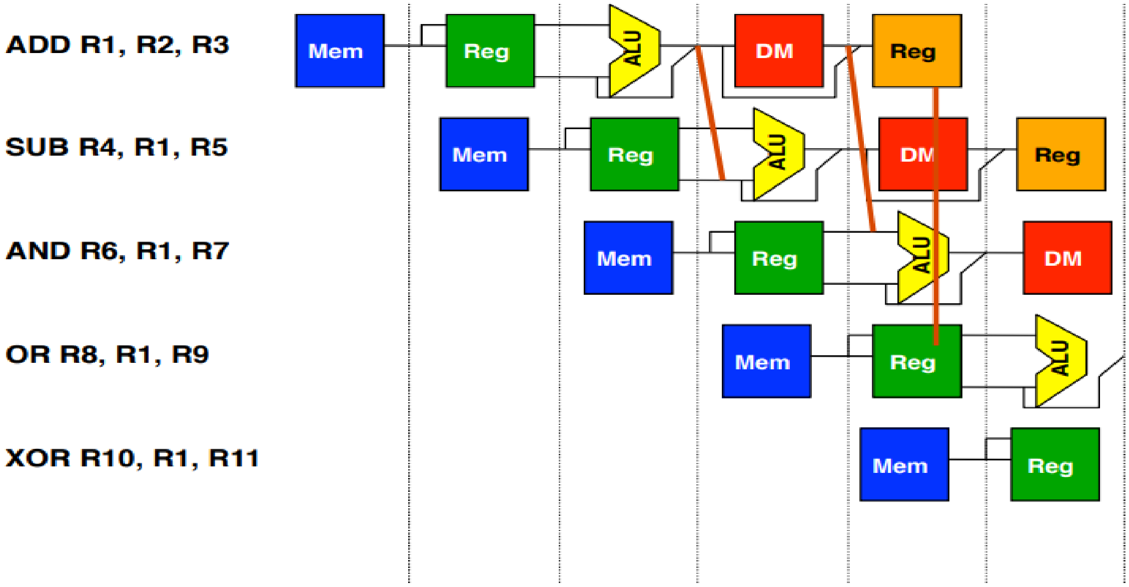
- ❖ The and instruction cannot fetch $2 until CC5
  - ✧ The and instruction remains in the IF/ID register until CC5
- ❖ Two bubbles are inserted into ID/EX at end of CC3 & CC4
  - ✧ Bubbles are NOP instructions: do not modify registers or memory
  - ✧ Bubbles delay instruction execution and waste clock cycles

Solution 2: Forwarding

Forwarding Also called bypassing: A method of resolving a data hazard by retrieving the data element from internal buffers rather than waiting for it to arrive from programmer visible registers or memory.

- **Generally speaking:**
  - **Forwarding occurs when a result is passed directly to functional unit that requires it.**
  - **Result goes from output of one unit to input of another**

When can we forward?



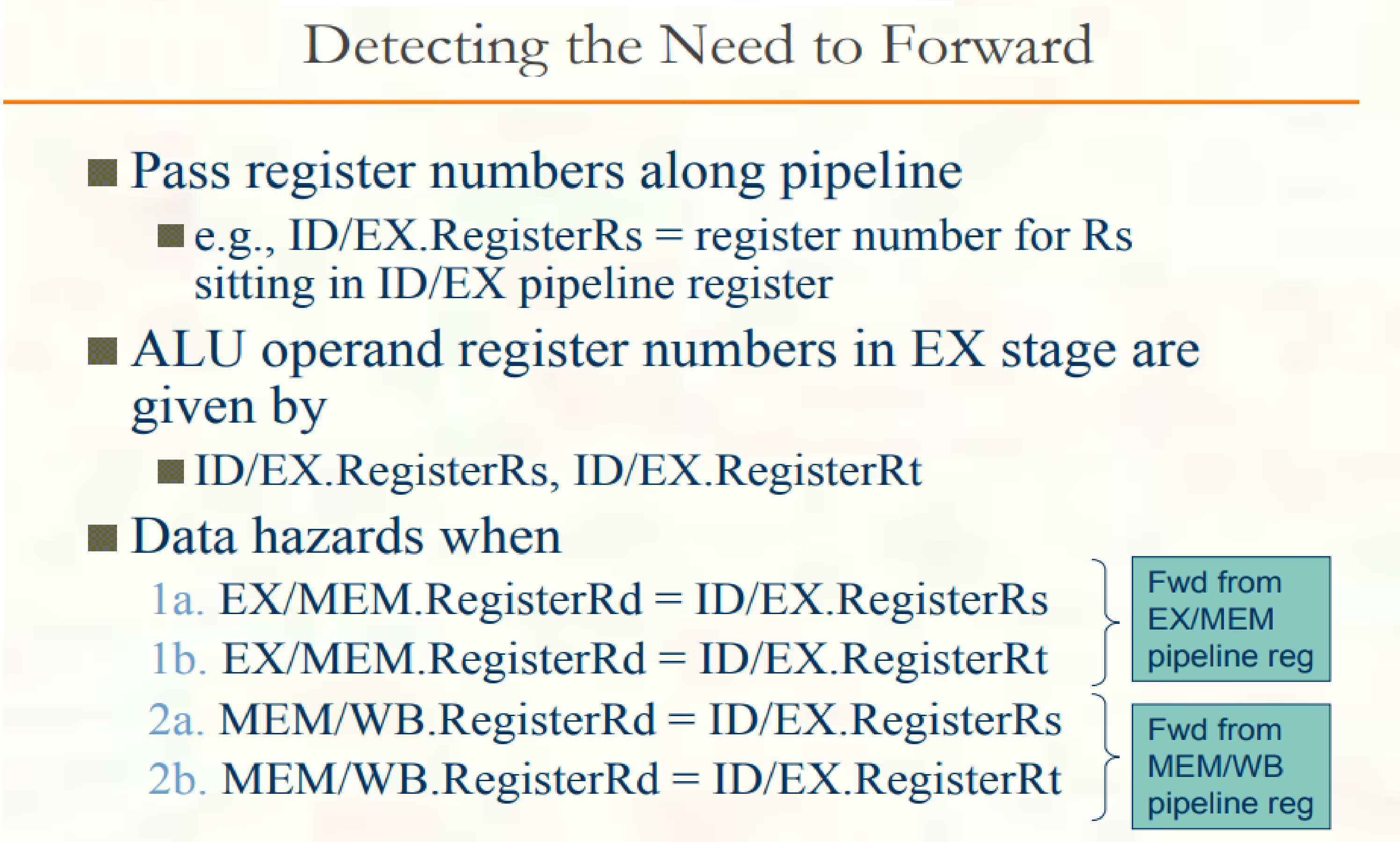
SUB gets info. from EX/MEM pipe register

AND gets info. from MEM/WB pipe register

OR gets info. by forwarding from register file

Time

Rule of thumb: **If line goes "forward" you can do forwarding.**
If its drawn backward, it's physically impossible.



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when

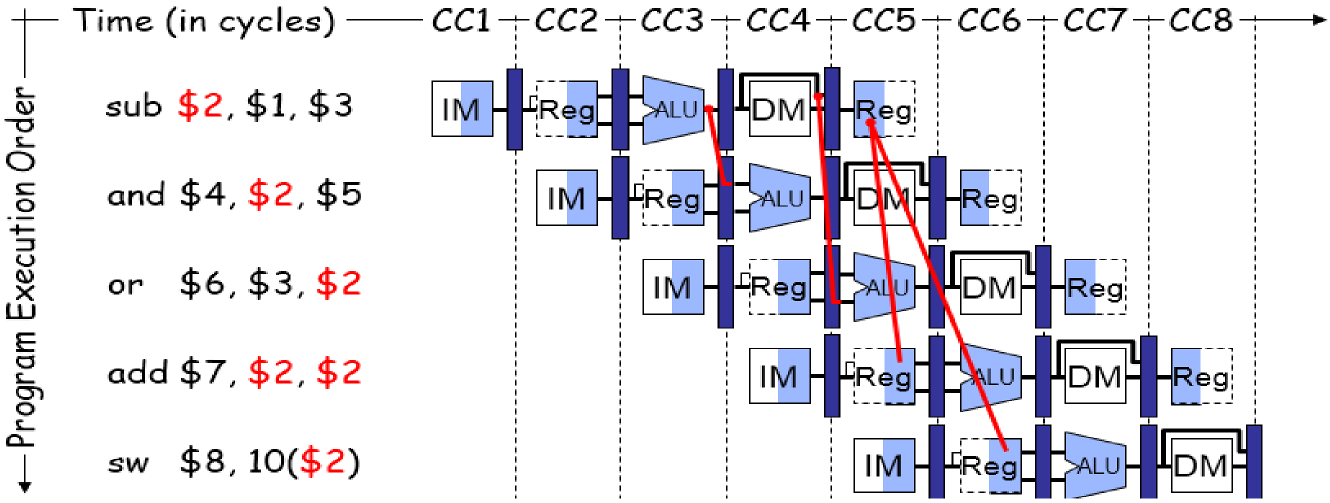
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

Solution 2: Forwarding ALU Result

- The ALU result is forwarded (fed back) to the ALU input
  - No bubbles are inserted into the pipeline and no cycles are wasted
- ALU result exists in either EX/MEM or MEM/WB register



Dependence Detection

- The sub-and is a first hazard:

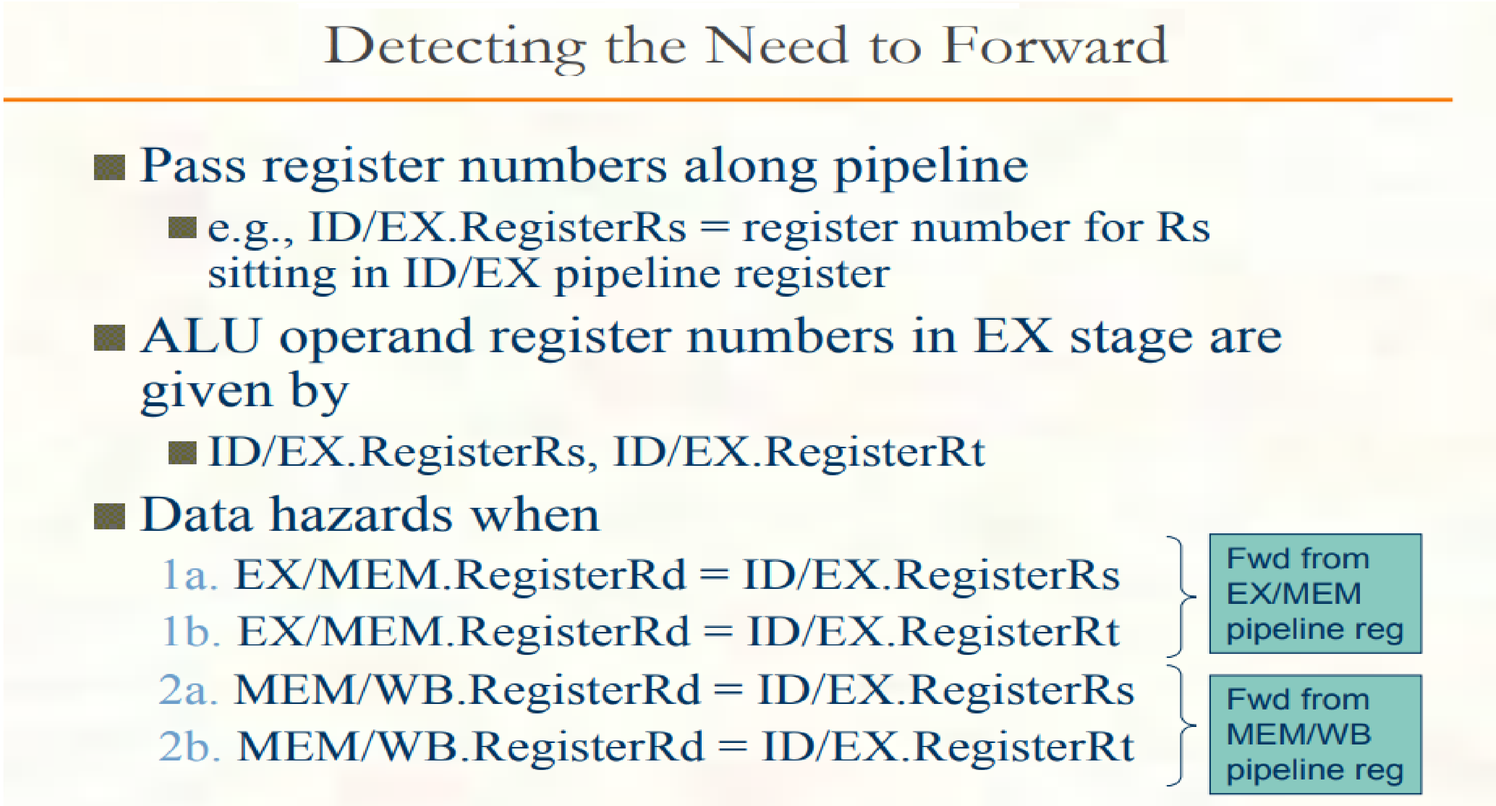
  EX/MEM.RegisterRd = ID/EX.RegisterRs = $2

- The sub-or is a second hazard:

  MEM/WB.RegisterRd = ID/EX.RegisterRt = $2

- The two dependences on sub-add are not hazards because the register file supplies the proper data during the ID stage of add.

- There is no data hazard between sub and sw because sw reads $2 the clock cycle after sub writes $2.



Detecting the Need to Forward

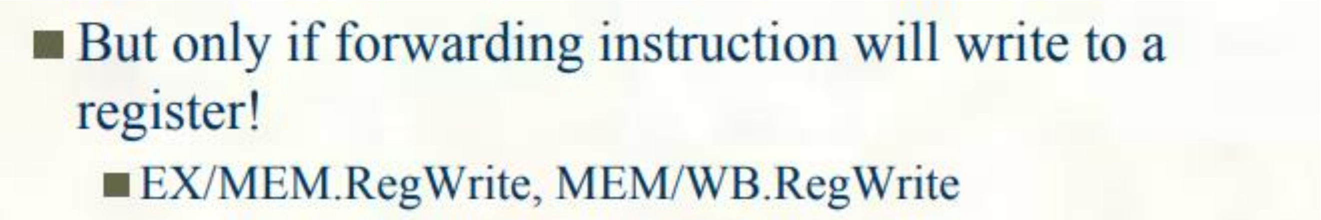
- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

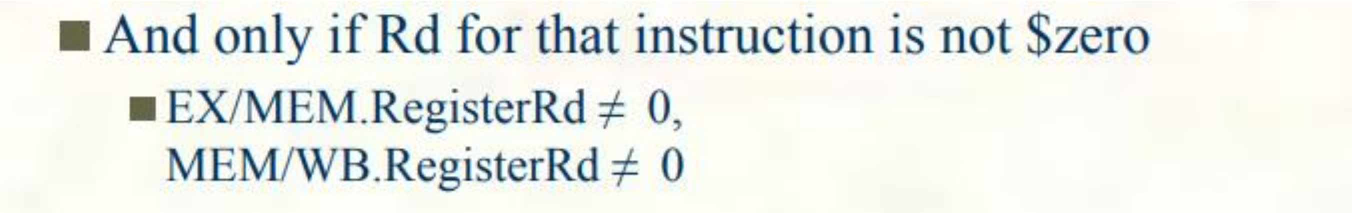
Fwd from MEM/WB pipeline reg

However, not all instructions perform register writes. So, we add the following requirement to our policy: **the RegWrite signal must be asserted in the WB control field during the EX stage for type 1 hazards and the MEM stage for type 2 hazards.**

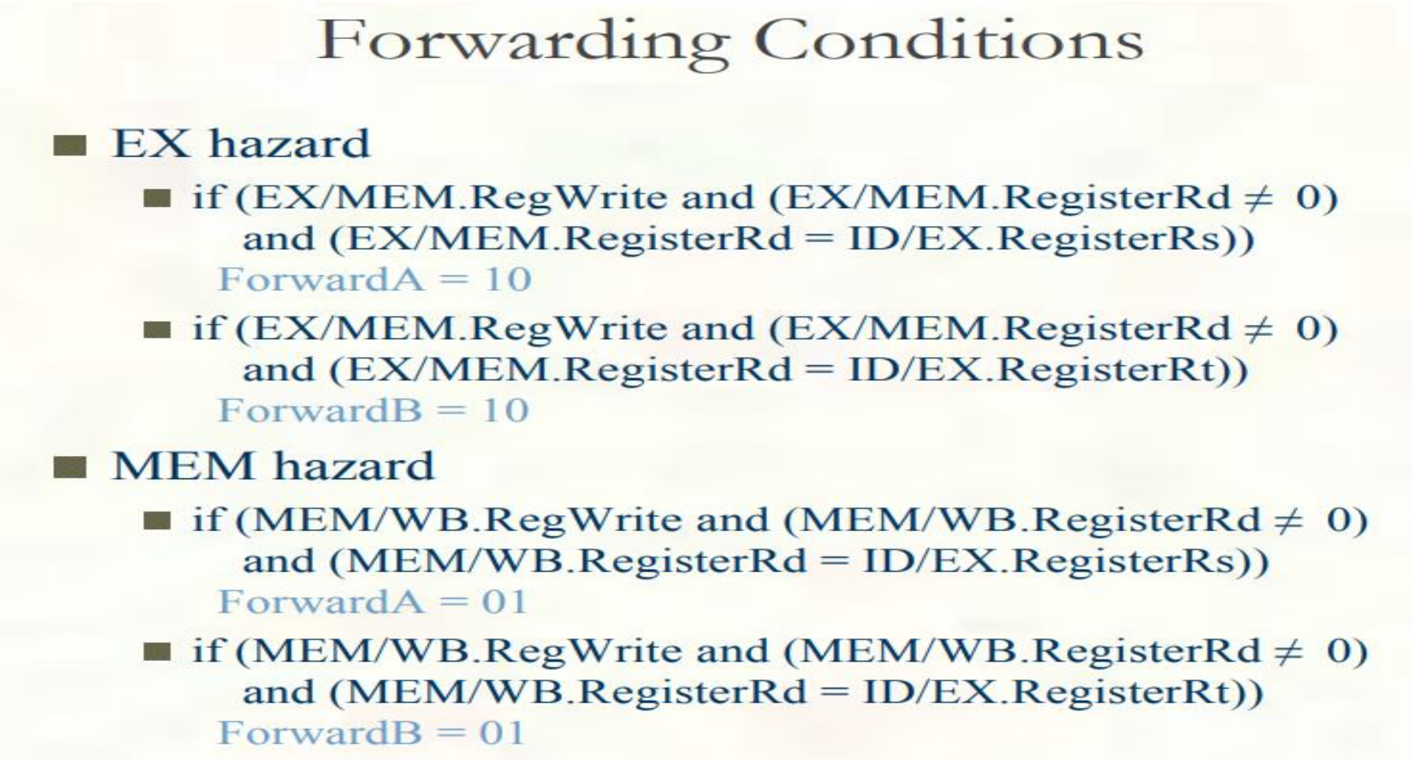


- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite

Also, we do not allow results to be written to the $0 register so, in the event that an instruction uses $0 as its destination (which is legal), we should not forward the result



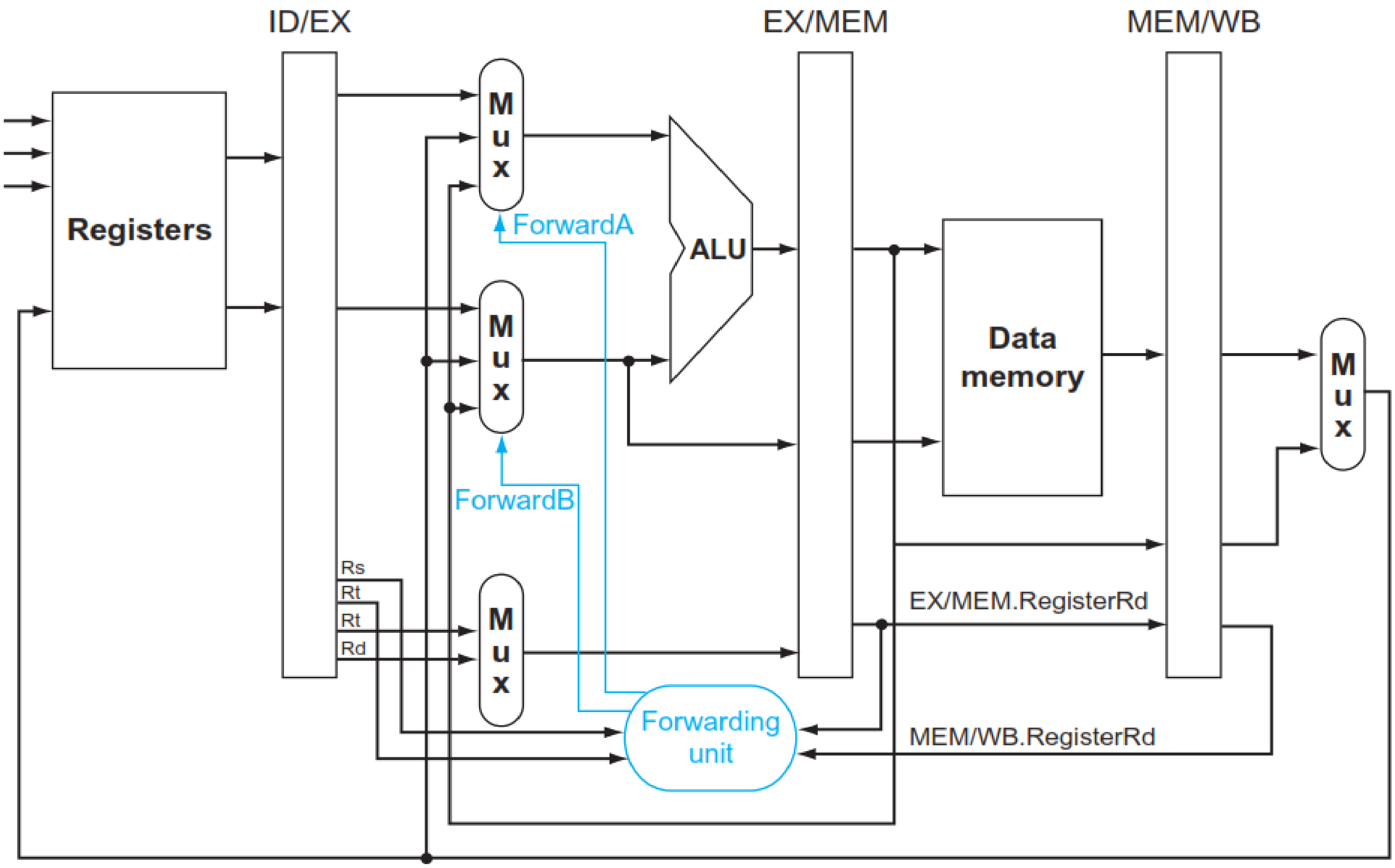
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0



Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

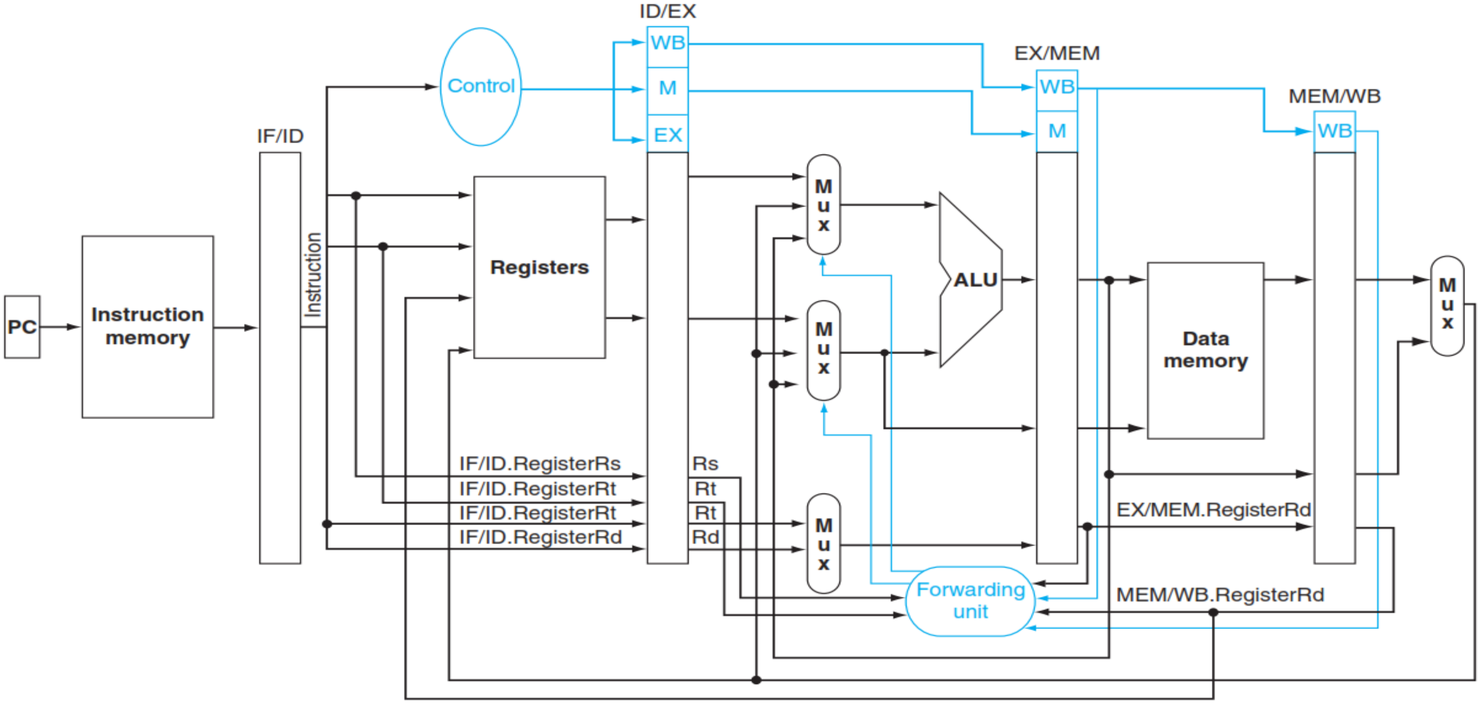
Implementing Forwarding



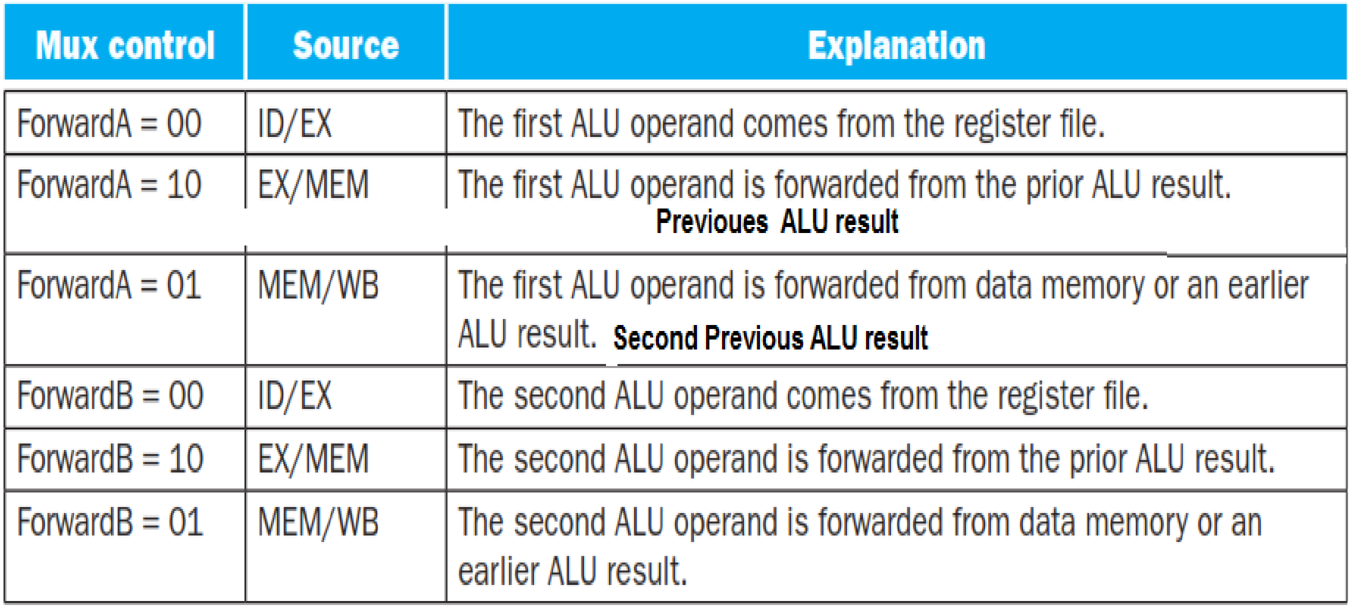
Pipelined Architecture with Forwarding

00: Register file to ALU

10: ALU to ALU



01: MEM Data or ALU to ALU



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result. **Previoues ALU result**
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result. **Second Previous ALU result**
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Forwarding Example

Instruction sequence:

```
lw   $4, 100($9)
add  $7, $5, $6
sub  $8, $4, $7
```

When `lw` reaches the MEM stage

`add` will be in the ALU stage

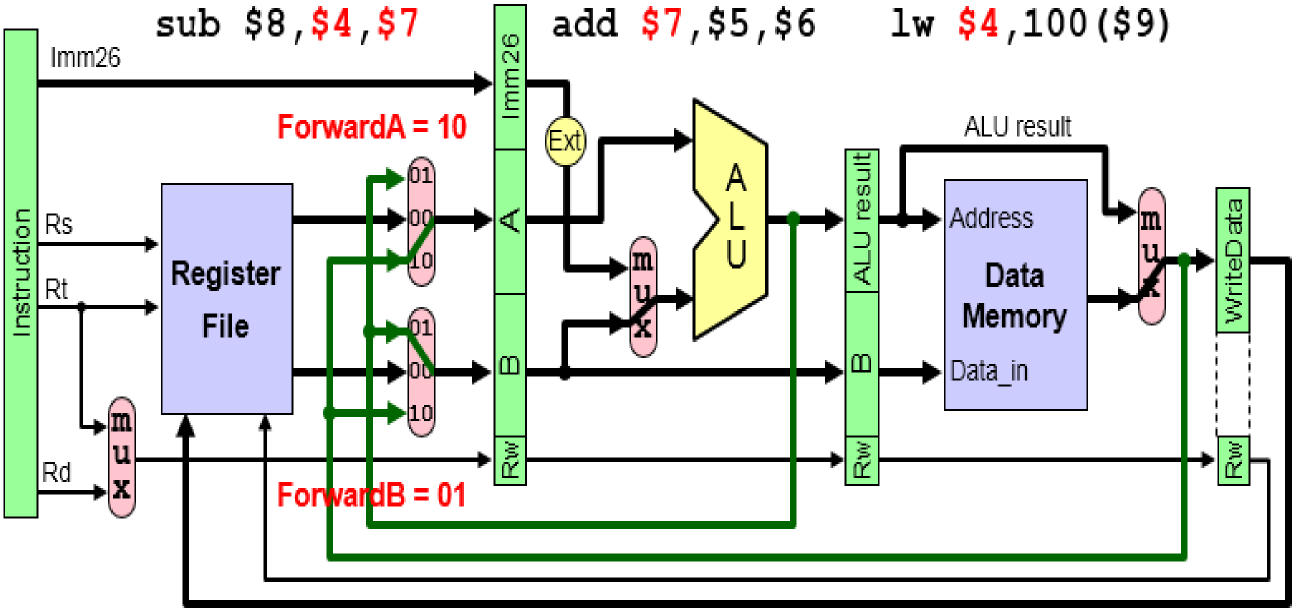
`sub` will be in the Decode stage

ForwardA = 10

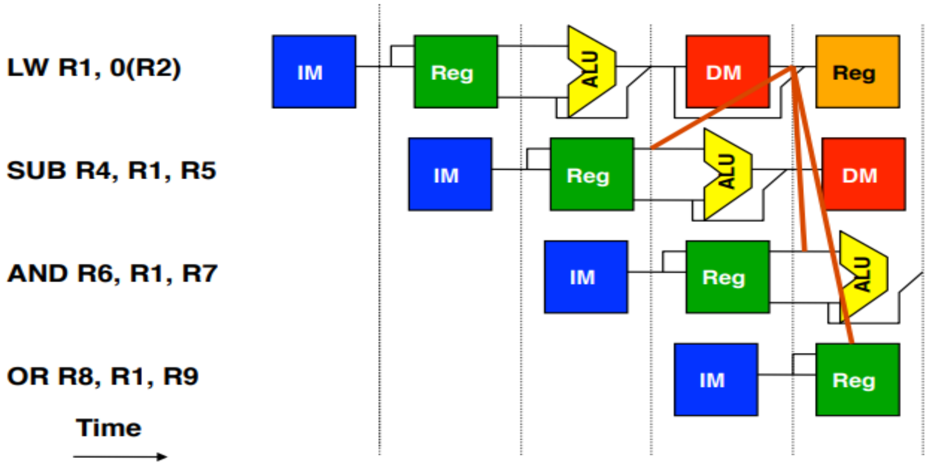
Forward data from MEM stage

ForwardB = 01

Forward ALU result from ALU stage



Forwarding doesn't always work



LW R1, 0(R2)

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

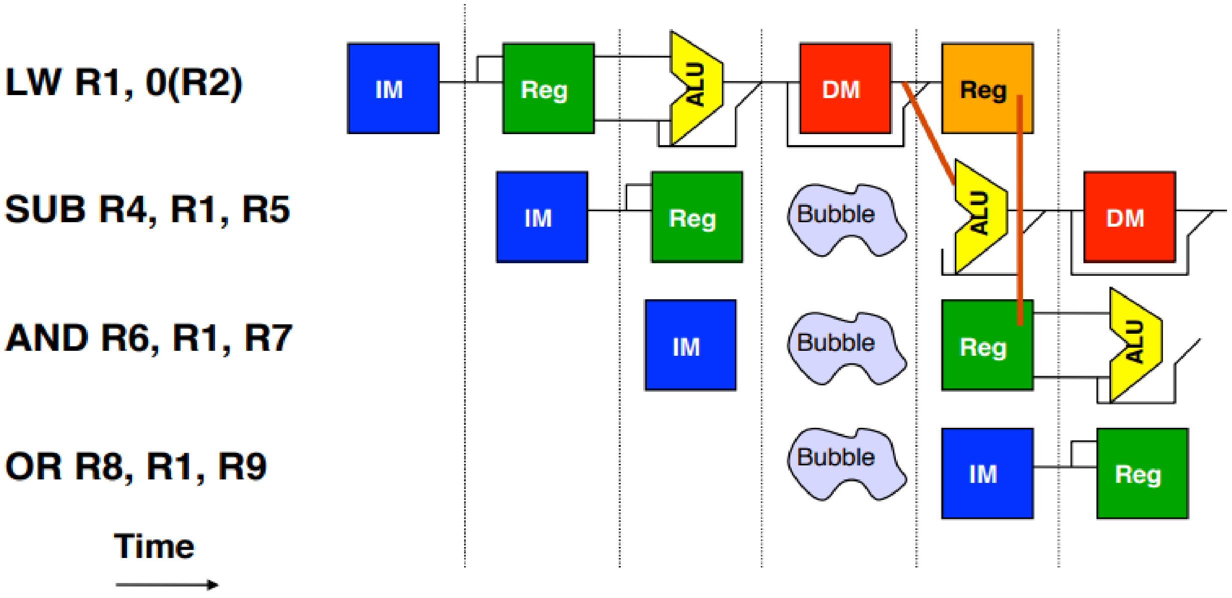
Time

Load has a latency that forwarding can't solve.

Pipeline must stall until hazard cleared (starting with instruction that wants to use data until source produces it).

Can't get data to subtract b/c result needed at beginning of CC #4, but not produced until end of CC #4.

The solution pictorially



LW R1, 0(R2)

SUB R4, R1, R5

AND R6, R1, R7

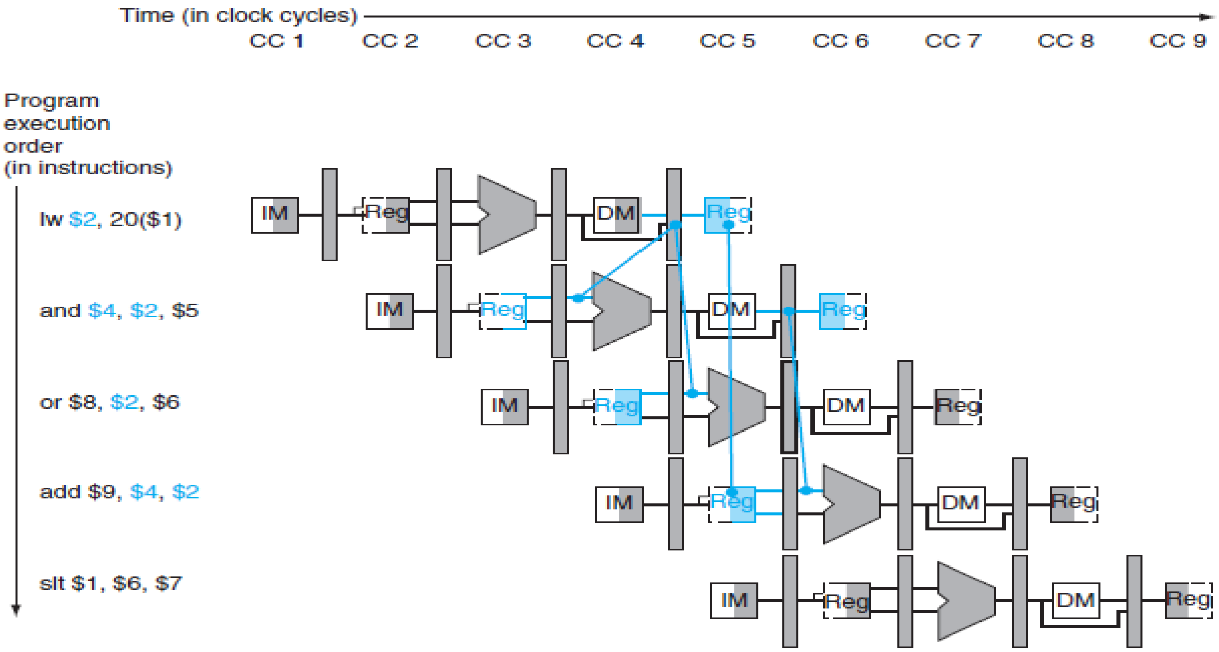
OR R8, R1, R9

Time

Insertion of bubble causes # of cycles to complete this sequence to grow by 1

Load Delay

- Unfortunately, not all data hazards can be forwarded
  - Load has a delay that cannot be eliminated by forwarding
- In the example shown below …
  - The LW instruction does not read data until end of CC4
  - Cannot forward data to AND at end of CC3 - NOT possible



- Detecting a RAW hazard after a Load instruction:
  - The load instruction will be in the EX stage
  - Instruction that depends on the load data is in the decode stage
- Condition for stalling the pipeline

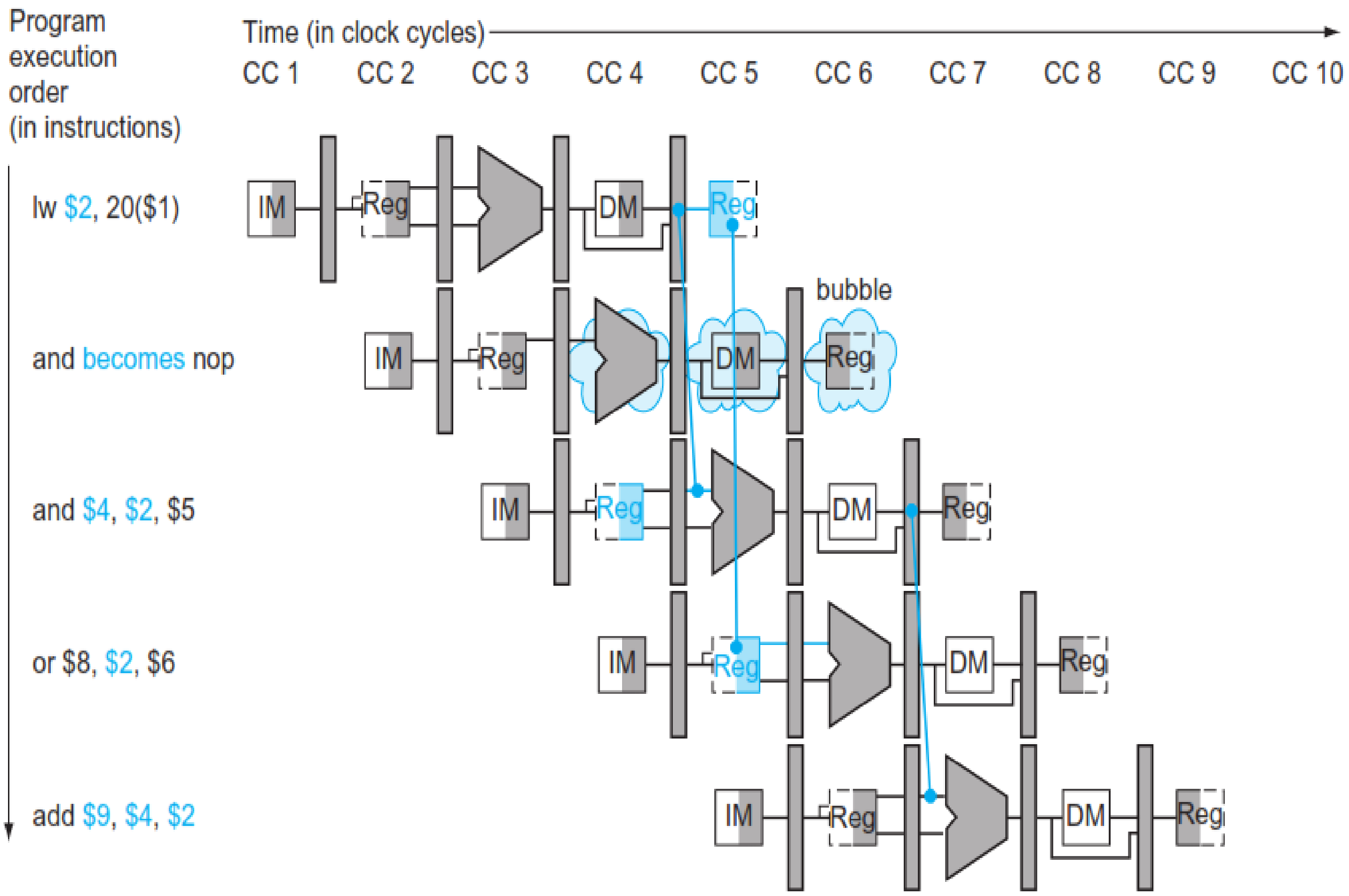
if (ID/EX.MemRead

and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or

(ID/EX.RegisterRt = IF/ID.RegisterRt)))

stall the pipeline

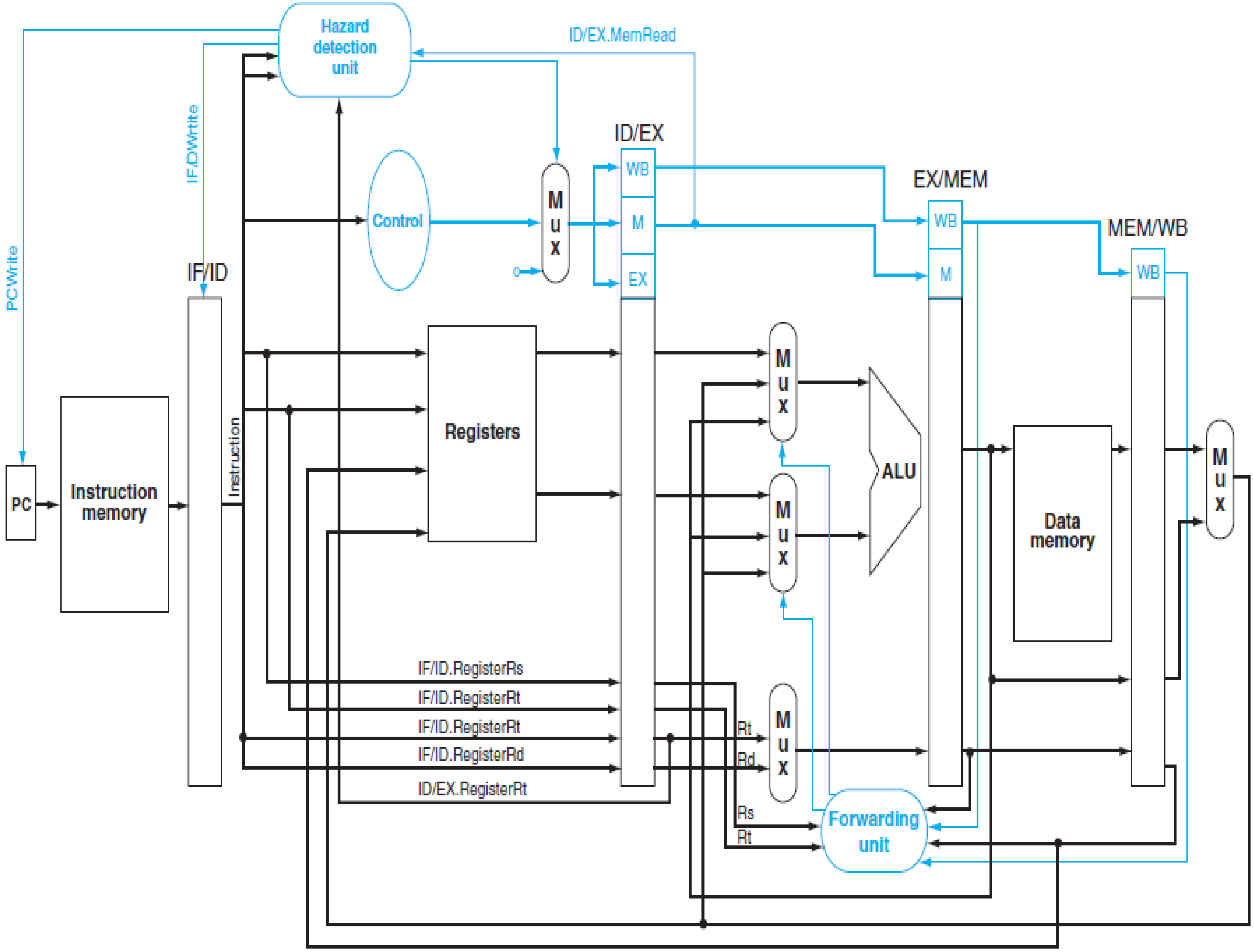
- The first line tests to see if the instruction is a load: the only instruction that reads data memory is a load. The next two lines check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage.
- If the condition holds, the instruction stalls one clock cycle. After this 1-cycle stall, the forwarding logic can handle the dependence and execution proceeds.



- A bubble is inserted beginning in clock cycle 4, by changing the **and** instruction to a **nop**. Note that the **and** instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4).

- Likewise the OR instruction is fetched in clock cycle 3, but its IF stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.



Write After Read – WAR Hazard

- Instruction J should write its result after it is read by I
- Called an anti-dependence by compiler writers

```
I:  sub  $4,  $1,  $3      # $1 is read
J:  add  $1,  $2,  $3      # $1 is written
```

- Results from reuse of the name $1
- Hazard occurs when J writes $1 before I reads it
- Cannot occur in our basic 5-stage pipeline because:
  - Reads are always in stage 2, and
  - Writes are always in stage 5
  - Instructions are processed in order

Write After Write – WAW Hazard

- Instruction J should write its result after instruction I
- Called an output-dependence in compiler terminology

```
I:  sub  $1,  $4,  $3  # $1 is written
J:  add  $1,  $2,  $3  # $1 is written again
```

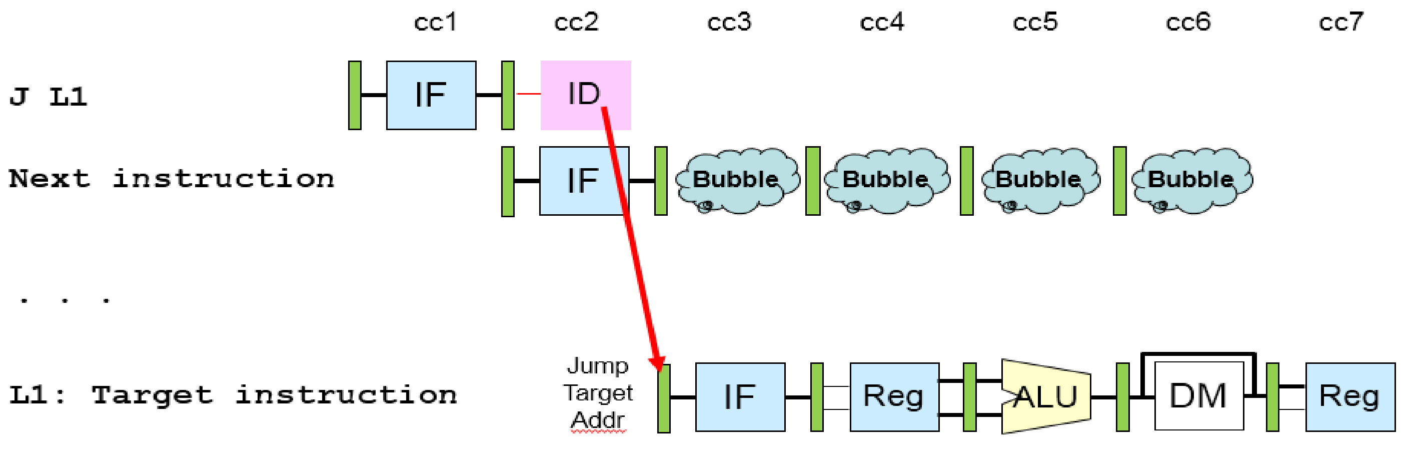
- This hazard also results from the reuse of name $1
- Hazard occurs when writes occur in the wrong order
- Can’t happen in our basic 5-stage pipeline because:
  - All writes are ordered and always take place in stage 5
- WAR and WAW hazards can occur in complex pipelines
- Notice that Read After Read – RAR is NOT a hazard

Control Hazards

- Jump and Branch can cause great performance loss
- Jump instruction needs only the jump target address
- Branch instruction needs two things:
  - Branch Result — Taken or Not Taken
  - Branch Target Address
    - PC + 4 — If Branch is NOT taken
    - PC + 4 + 4 × immediate — If Branch is Taken
- Jump and Branch targets are computed in the ID stage
  - At which point a new instruction is already being fetched
  - Jump Instruction: 1-cycle delay
  - Branch: 2-cycle delay for branch result (taken or not taken)

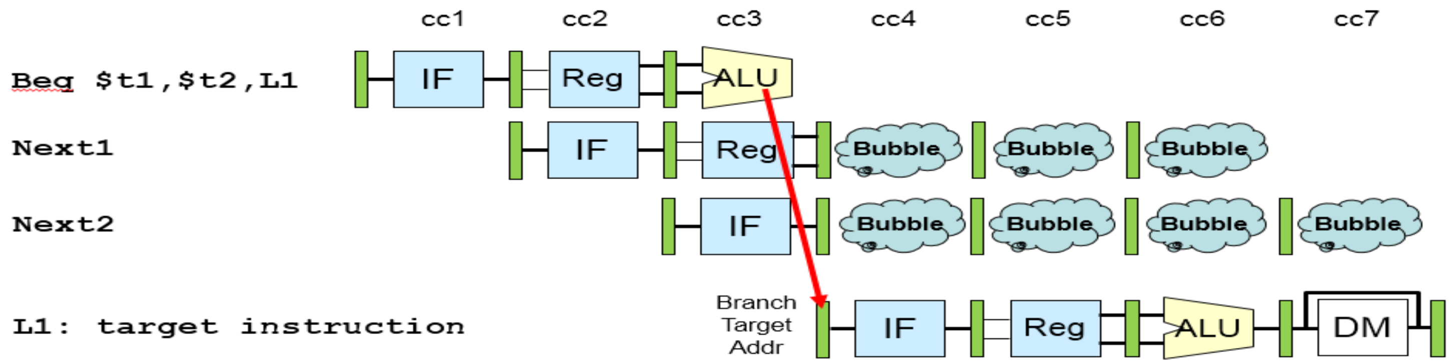
1-Cycle Jump Delay

- Control logic detects a Jump instruction in the 2nd Stage
- Next instruction is fetched anyway
- Convert Next instruction into bubble (Jump is always taken)



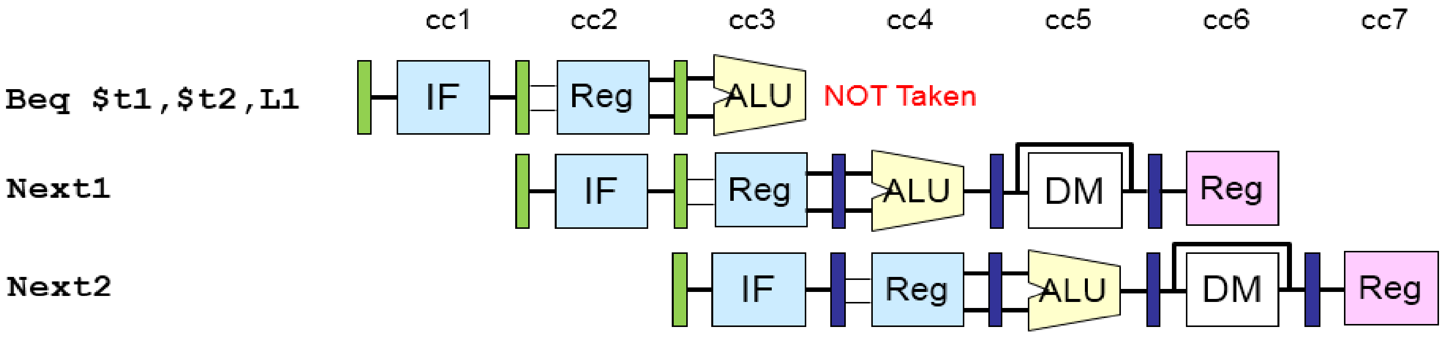
2-Cycle Branch Delay

- Control logic detects a Branch instruction in the 2nd Stage
- ALU computes the Branch outcome in the 3rd Stage
- Next1 and Next2 instructions will be fetched anyway
- Convert Next1 and Next2 into bubbles if branch is taken



Predict Branch NOT Taken

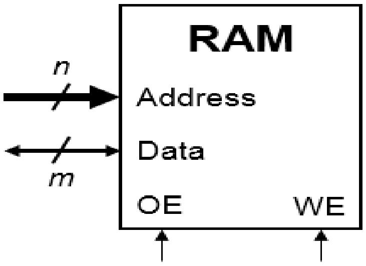
- Branches can be predicted to be NOT taken
- If branch outcome is NOT taken then
  - Next1 and Next2 instructions can be executed
  - Do not convert Next1 & Next2 into bubbles
  - No wasted cycles



Memory Hierarchy and Caches

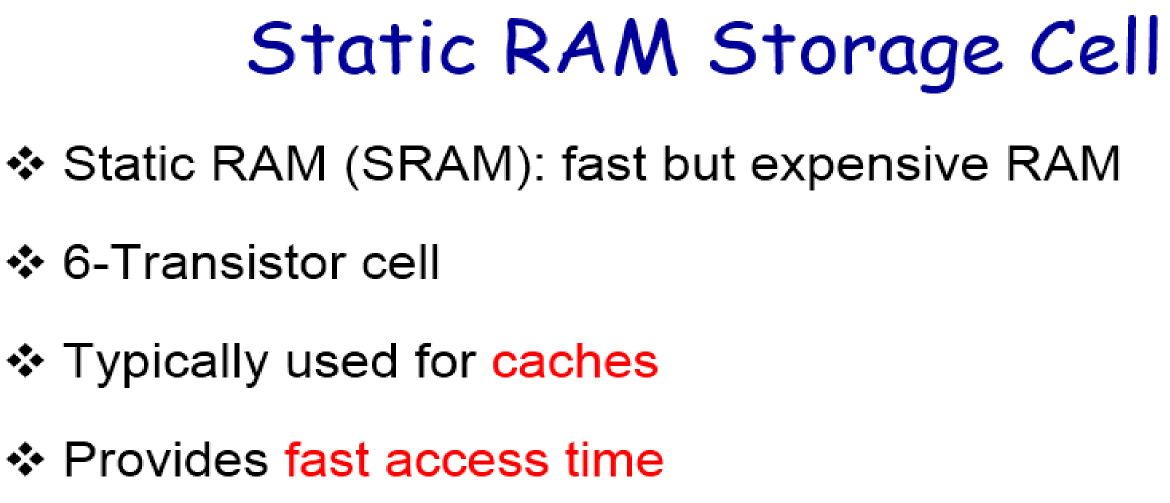
Random Access Memory

- Large arrays of storage cells
- Volatile memory
  - Hold the stored data as long as it is powered on
- Random Access
  - Access time is practically the same to any data on a RAM chip
- Output Enable (OE) control signal
  - Specifies read operation
- Write Enable (WE) control signal
  - Specifies write operation
- $2^n \times m$ RAM chip: n-bit address and m-bit data



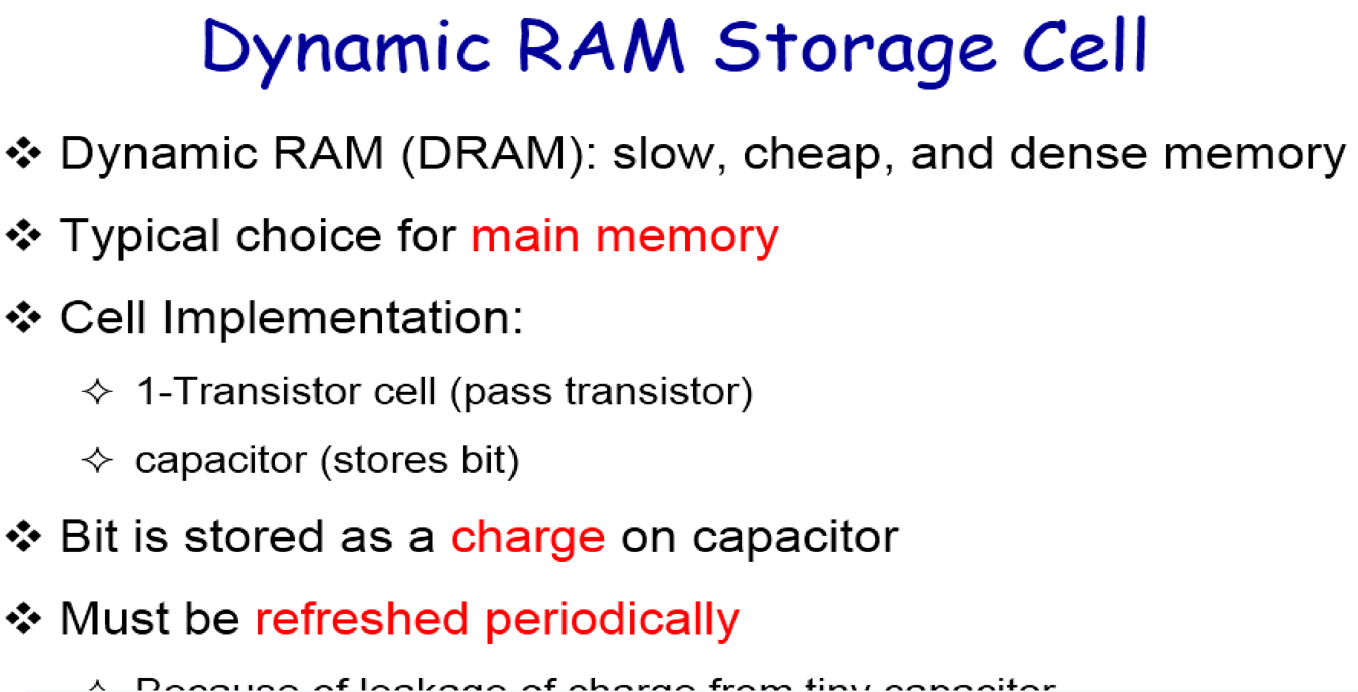
Memory Technology

- Static RAM (SRAM) for Cache
  - Requires 6 transistors per bit
  - Requires low power to retain bit
- Dynamic RAM (DRAM) for Main Memory
  - One transistor + capacitor per bit
  - Must be re-written after being read
  - Must also be periodically refreshed
    - Each row can be refreshed simultaneously
  - Address lines are multiplexed
    - Upper half of address: Row Access Strobe (RAS)
    - Lower half of address: Column Access Strobe (CAS)



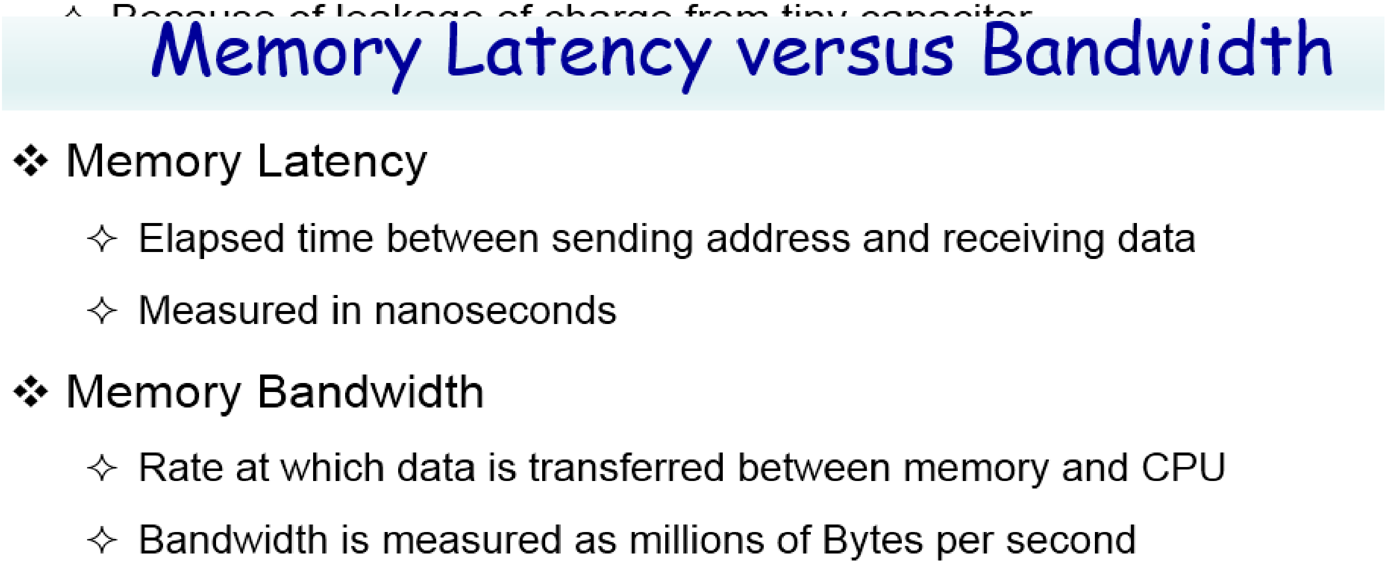
Static RAM Storage Cell

- Static RAM (SRAM): fast but expensive RAM
- 6-Transistor cell
- Typically used for caches
- Provides fast access time



Dynamic RAM Storage Cell

- Dynamic RAM (DRAM): slow, cheap, and dense memory
- Typical choice for main memory
- Cell Implementation:
  - 1-Transistor cell (pass transistor)
  - capacitor (stores bit)
- Bit is stored as a charge on capacitor
- Must be refreshed periodically
  - Because of leakage of charge from tiny capacitor

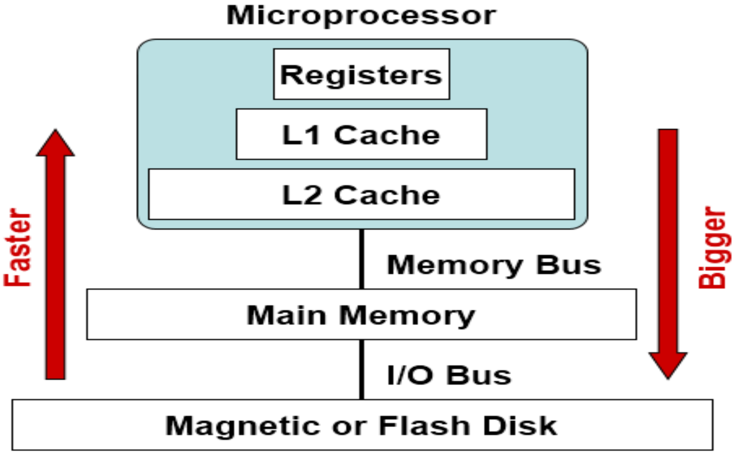


Memory Latency versus Bandwidth

- Memory Latency
  - Elapsed time between sending address and receiving data
  - Measured in nanoseconds
- Memory Bandwidth
  - Rate at which data is transferred between memory and CPU
  - Bandwidth is measured as millions of Bytes per second

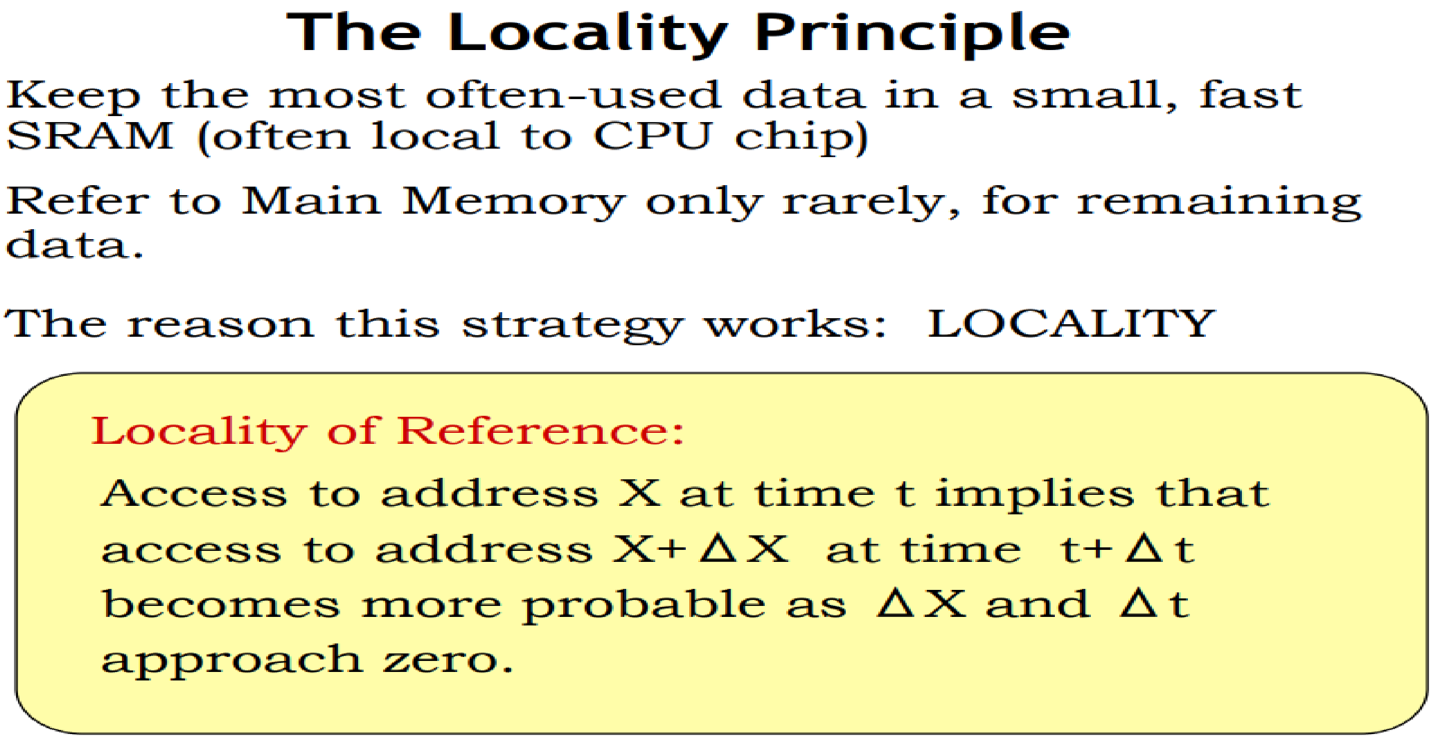
Typical Memory Hierarchy

- Registers are at the top of the hierarchy
  - Typical size < 1 KB
  - Access time < 0.5 ns
- Level 1 Cache (8 – 64 KB)
  - Access time: 1 ns
- L2 Cache (512KB – 8MB)
  - Access time: 3 – 10 ns
- Main Memory (4 – 16 GB)
  - Access time: 50 – 100 ns
- Disk Storage (> 200 GB)
  - Access time: 5 – 10 ms



The Need for Cache Memory

- Widening speed gap between CPU and main memory
  - Processor operation takes less than 1 ns
  - Main memory requires about 100 ns to access
- Each instruction involves at least one memory access
  - One memory access to fetch the instruction
  - A second memory access for load and store instructions
- Cache memory can help bridge the CPU-memory gap
- Cache memory is small in size but fast



The Locality Principle

Keep the most often-used data in a small, fast SRAM (often local to CPU chip)

Refer to Main Memory only rarely, for remaining data.

The reason this strategy works: LOCALITY

Locality of Reference:

Access to address X at time t implies that access to address X+ΔX at time t+Δt becomes more probable as ΔX and Δt approach zero.

There are two different types of locality:

Temporal locality (*locality in time*): **The principle stating that if a data location is referenced then it will tend to be referenced again soon.**

❖ Caches exploit temporal locality by …

✧ Keeping recently accessed data closer to the processor

Spatial locality (*locality in space*): **The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.**

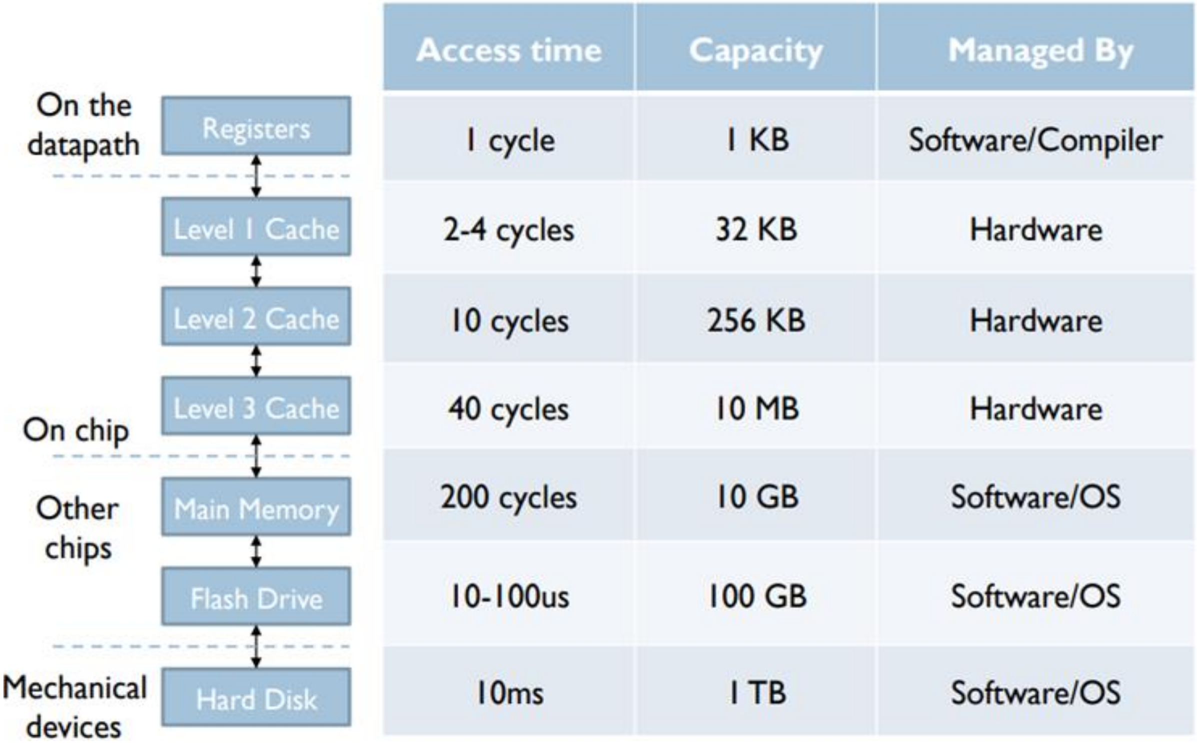
❖ Caches exploit spatial locality by …

✧ Moving blocks consisting of multiple contiguous words

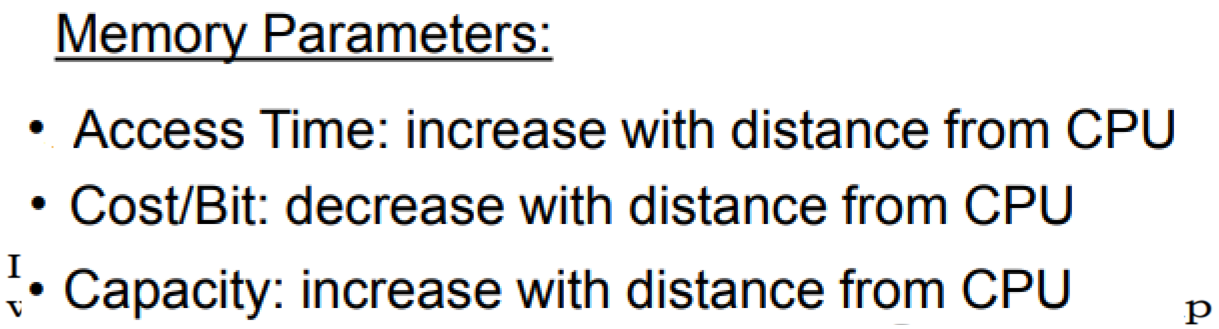
➢ *We take advantage of the principle of locality by implementing the memory of a computer as a memory hierarchy. A memory hierarchy consists of multiple levels of memory with different speeds and* *sizes. The faster memories are more expensive per bit than the slower memories and thus are smaller.*

Memory Hierarchy

A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase.



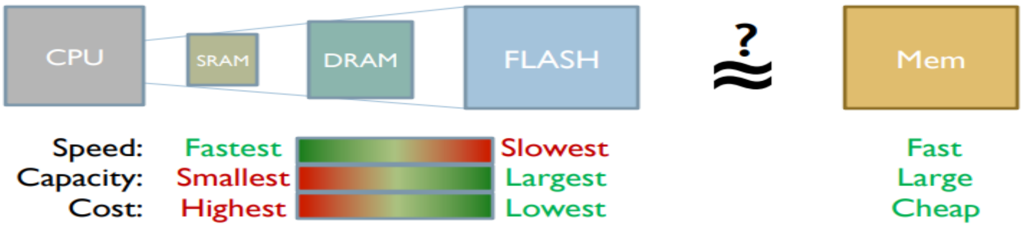
Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS



Memory Parameters:

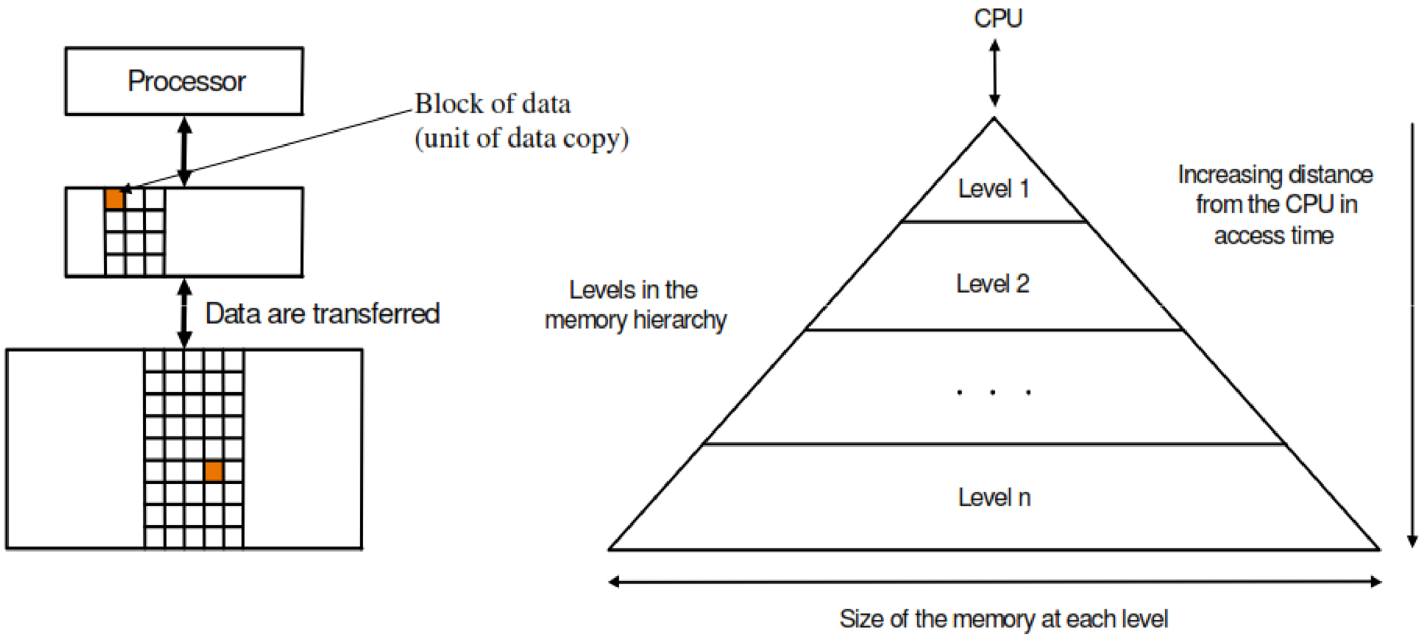
- Access Time: increase with distance from CPU
- Cost/Bit: decrease with distance from CPU
- Capacity: increase with distance from CPU

I
v
p
memory?

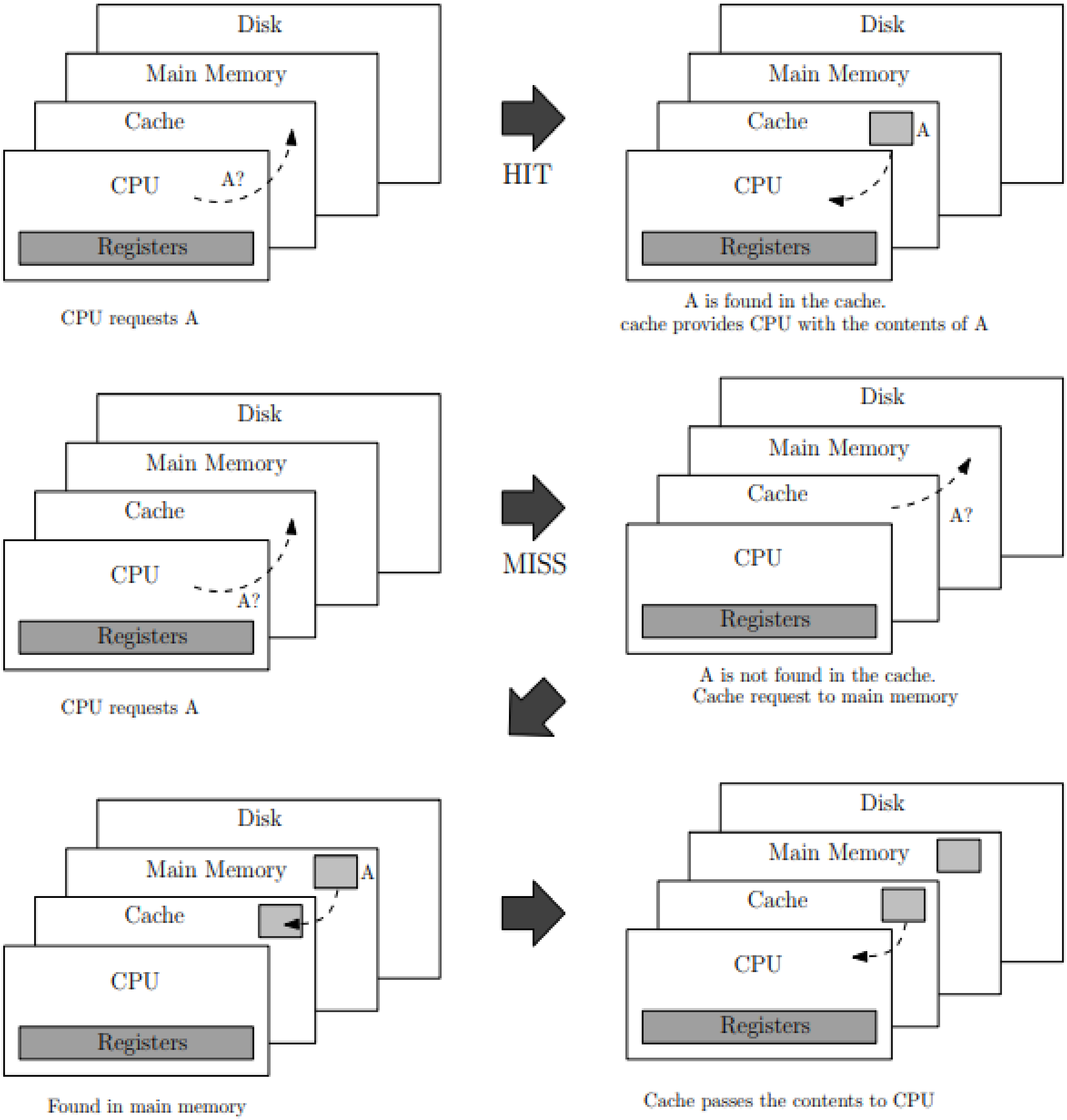


Memory Hierarchy Levels

- Hierarchy is inclusive, every level is subset of lower level



- ***Block***: minimum unit of data to move between levels
- If accessed data is present in upper level (closer to CPU)
  - ✓ ***hit***: data requested is in upper level
  - ✓ ***hit time***: time to access and deliver the data from the upper level
  - ✓ ***hit ratio***: percentage of time the data is found in the upper level (hits/accesses)
- If accessed data is absent
  - ✓ ***miss***: data requested is not in upper level
  - ✓ i.e. a block copied from lower level (farther from CPU)
  - ✓ ***miss penalty***: time to access and copy data from lower level to upper level, then to CPU
  - ✓ ***miss ratio***: : percentage of time the data is not hits.
    - miss ratio = 1 – hit ratio



CPU requests A

A is found in the cache.
cache provides CPU with the contents of A

CPU requests A

A is not found in the cache.
Cache request to main memory

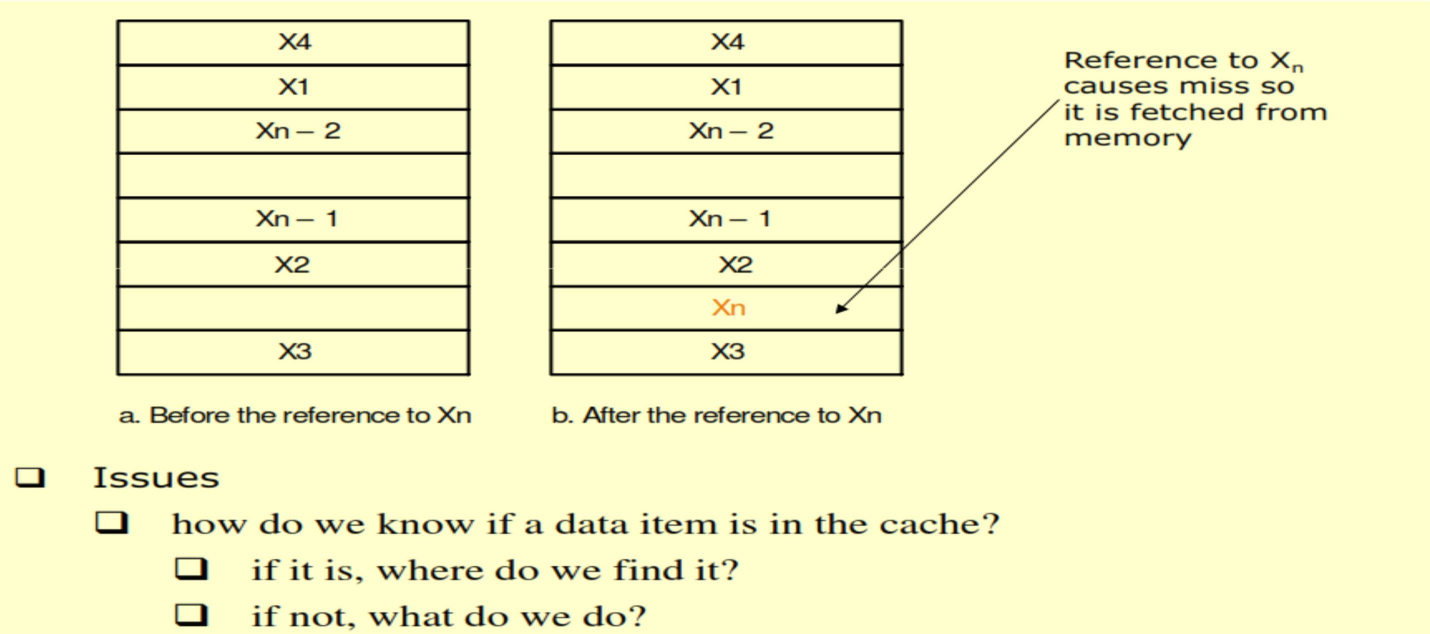
Found in main memory
Contents are copied to cache

Cache passes the contents to CPU

Memory access, resulting in a hit or a miss

A **hit** occurs if the data required by the processor appears in some block in the upper level and a **miss** occurs if this is not the case and the lower level needs to be accessed to copy the block that contains the data requested by the CPU into the upper level.

- In the following figure, the cache contains a collection of recent references X_1, X_2, …, X_{n-1} , and the processor requests a word X_n that is not in the cache. This request results in a **miss**, and the word X_n is brought from memory into the cache.



Reference to X_n causes miss so it is fetched from memory

a. Before the reference to Xn

b. After the reference to Xn

- ❑ Issues
  - ❑ how do we know if a data item is in the cache?
    - ❑ if it is, where do we find it?
    - ❑ if not, what do we do?

- The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the **address** of the word in memory.

- This cache structure is called **direct mapped,** since each memory location is mapped directly to exactly one location in the cache.

- **direct-mapped cache** : A cache structure in which each memory location is mapped to exactly one location in the cache.

- almost all direct-mapped caches use this mapping to find a block:

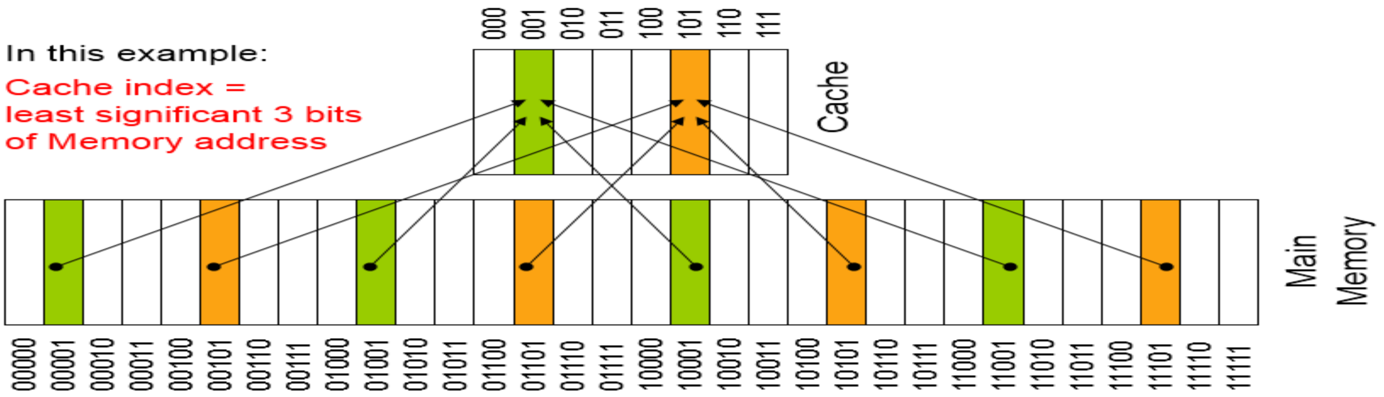
(Block address in main mem.) MOD (Number of blocks in the cache)

- In fact, this equation can be implemented in a very simple way if the number of blocks in the cache is a power of two, 2^x , since

(Block address in main mem.) MOD 2^x = x lower-order bits of the block address

Block Placement: Direct Mapped

- **Block**: unit of data transfer between cache and memory
- **Direct Mapped Cache**:
  - A block can be placed in exactly one location in the cache



In this example:
Cache index =
least significant 3 bits
of Memory address

Drawback: may overwrite some parts of cache while other parts are empty

A given memory block can be mapped into one and only cache line. Here is an example of mapping

Cache line	Main memory block
0	0, 8, 16, 24, ... 8n
1	1, 9, 17. 25, ... 8n+1
2	2, 10, 18, 26, ... 8n+2
3	3, 11, 19, 27, ... 8n+3

Advantage

No need of expensive associative search!

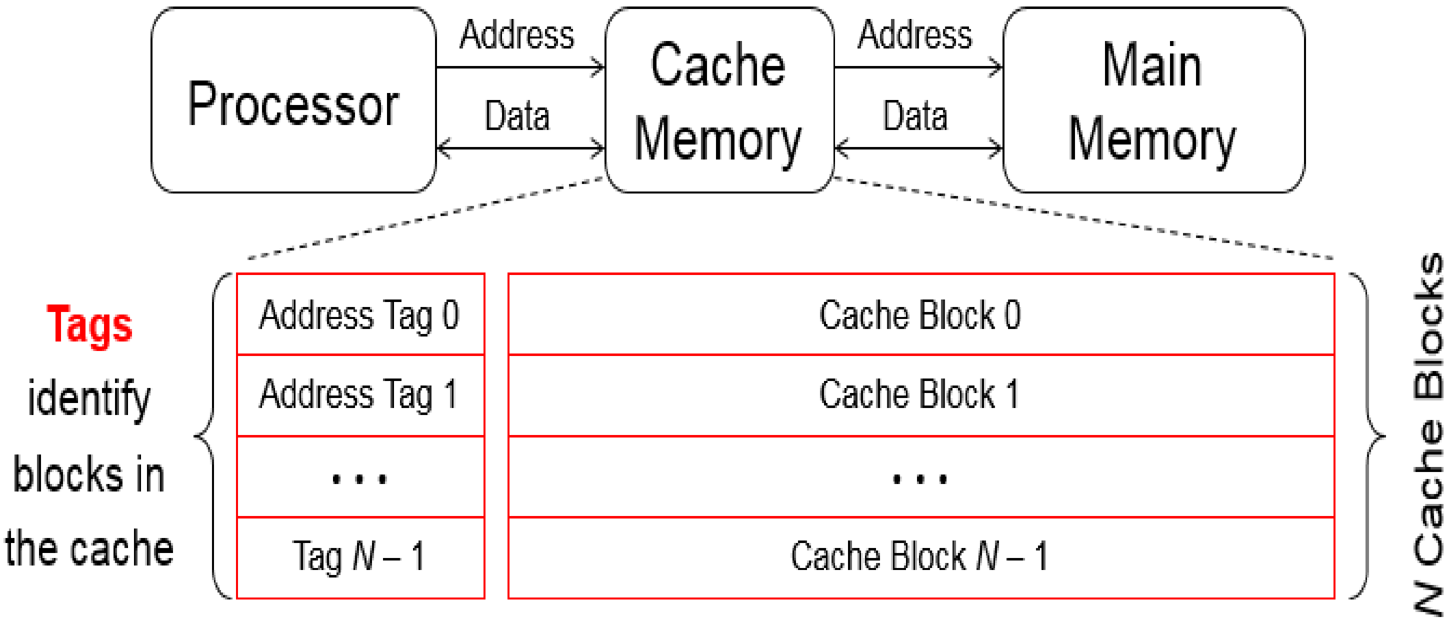
Disadvantage

Miss rate may go up due to possible increase of mapping conflicts.

- Since each block in the cache can contain the contents of different memory locations that have the same **x least-significant address bits**, every block in the cache is augmented with a **tag field**. The tag bits allow to uniquely identify which memory content is stored in a given block of the cache.

- ❑ Location determined by address: *direct mapped*
  - ✓ cache block address = memory block address *mod* cache size (*unique*)
  - ✓ if cache size = 2^m, cache address = lower m bits of n-bit memory address
  - ✓ remaining upper n-m bits kept as ***tag*** *bits* at each cache block
  - ✓ also need a ***valid*** *bit* to recognize valid entry

Inside a Cache Memory



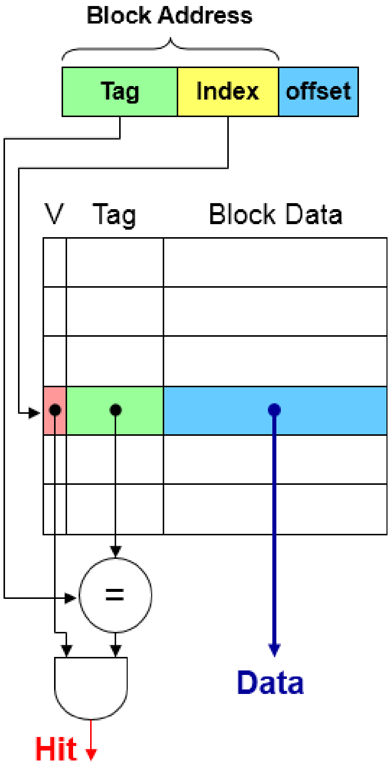
Address Tag 0	Cache Block 0
Address Tag 1	Cache Block 1
• • •	• • •
Tag *N* – 1	Cache Block *N* – 1

Direct-Mapped Cache

- A memory address is divided into
  - **Block address**: identifies block in memory
  - **Block offset**: to access bytes within a block
- A block address is further divided into
  - **Index**: used for direct cache access
  - **Tag**: most-significant bits of block address

  Index = Block Address **mod** *Cache Blocks*
- Tag must be stored also inside cache
  - For block identification
- A **valid bit** is also required to indicate
  - Whether a cache block is valid or not

  What if there is no data in a location?
  - **Valid bit**: **1** = present, **0** = not present , Initially 0

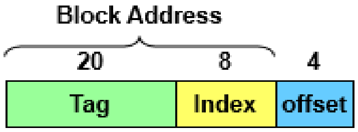


- **Cache hit**: block is stored inside cache
  - Index is used to access cache block
  - Address tag is compared against stored tag
  - If equal and cache block is valid then **hit**
  - Otherwise: **cache miss**
- If number of cache blocks is 2^n
  - n bits are used for the cache index
- If number of bytes in a block is 2^b
  - b bits are used for the block offset
- If 32 bits are used for an address
  - $32 - n - b$ bits are used for the tag
- Cache data size = 2^{n+b} bytes

❖ Example

- ✧ Consider a direct-mapped cache with 256 blocks
- ✧ Block size = 16 bytes
- ✧ Compute tag, index, and byte offset of address: 0x01FFF8AC

❖ Solution



- ✧ 32-bit address is divided into:
  - 4-bit byte offset field, because block size = 2^4 = 16 bytes
  - 8-bit cache index, because there are 2^8 = 256 blocks in cache
  - 20-bit tag field
- ✧ Byte offset = 0xC = 12 (least significant 4 bits of address)
- ✧ Cache index = 0x8A = 138 (next lower 8 bits of address)
- ✧ Tag = 0x01FFF (upper 20 bits of address)

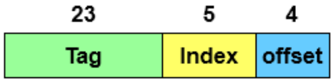
Example

❖ Consider a small direct-mapped cache with 32 blocks

- ✧ Cache is initially empty, Block size = 16 bytes
- ✧ The following memory addresses (in decimal) are referenced:

  1000, 1004, 1008, 2548, 2552, 2556.
- ✧ Map addresses to cache blocks and indicate whether hit or miss

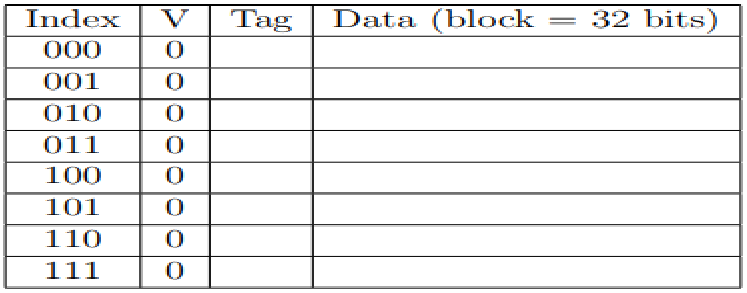
❖ Solution:



✧ 1000 = 0x3E8	cache index = 0x1E	Miss (first access)
✧ 1004 = 0x3EC	cache index = 0x1E	Hit
✧ 1008 = 0x3F0	cache index = 0x1F	Miss (first access)
✧ 2548 = 0x9F4	cache index = 0x1F	Miss (different tag)
✧ 2552 = 0x9F8	cache index = 0x1F	Hit
✧ 2556 = 0x9FC	cache index = 0x1F	Hit

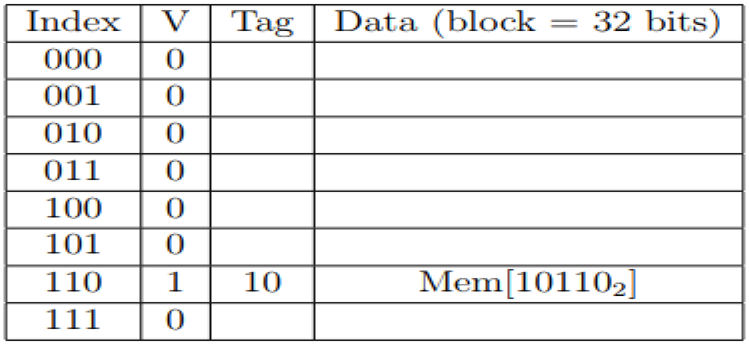
Example

At power-up, every cache line is invalid (V=0). Let’s consider the following sequence of memory references: 10110_2, 11010_2, 10110_2, 10000_2, 10010_2.



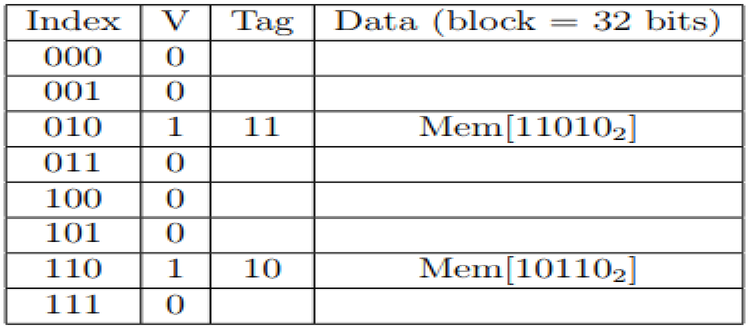
Index	V	Tag	Data (block = 32 bits)
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

- **For the first memory access, at 10110_2**, the 3 LSB, to index the cache, are **110**. The corresponding block in the cache is invalid **(V = 0)**, so we have a cache **miss**. The block containing the requested word is copied into the cache from the next **level below in the memory hierarchy**, the tag bits are set to **10** and the **valid bit** is **set** (as the cache block is now valid), resulting in the following state of the cache.



Index	V	Tag	Data (block = 32 bits)
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	1	10	Mem[10110_2]
111	0		

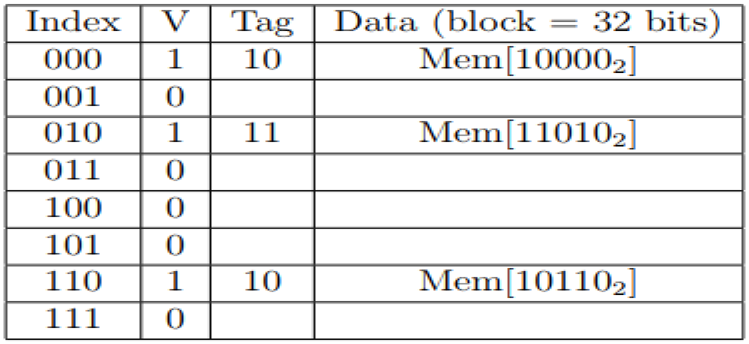
- **The next access is at word address 11010_2**. The index bits are 010. The corresponding block in the cache is invalid again, so we have a cache **miss**, copy the appropriate block from main memory, set the **tag bits to 11** and the **valid** bit to 1, resulting in the cache state below.



Index	V	Tag	Data (block = 32 bits)
000	0		
001	0		
010	1	11	Mem[11010_2]
011	0		
100	0		
101	0		
110	1	10	Mem[10110_2]
111	0		

- **The next access is at word address 10110_2**. The index bits are **110**. The corresponding block of the cache is valid (V = 1), with tag bits 10, which match the tag bits of the word address 10110_2. This implies a cache **hit**, so the cache can provide the CPU promptly with the requested data, Mem[10110_2].

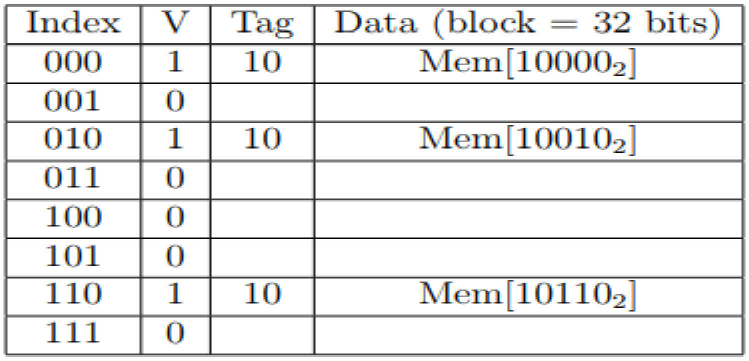
- **The next access is at word address 10000_2**. The index bits are **000 ,** which corresponds to an invalid cache block and thus a **miss**. Copying the right block from main memory into the cache and adjusting tag and valid bit results in the following state of the cache.



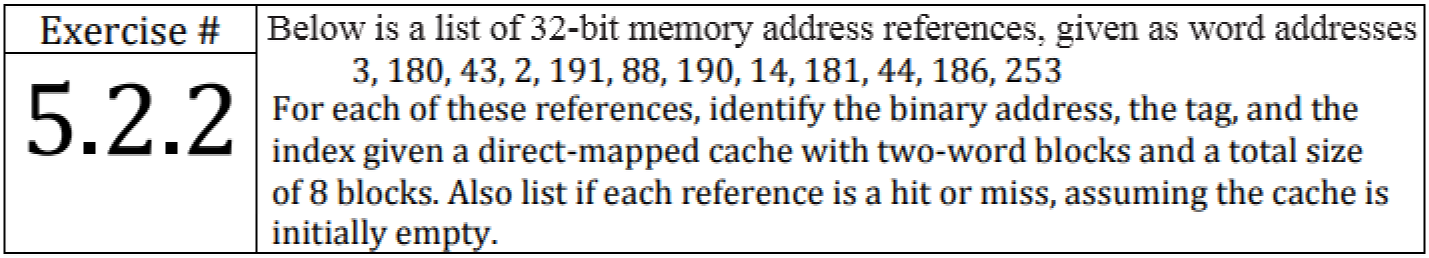
Index	V	Tag	Data (block = 32 bits)
000	1	10	Mem[10000_2]
001	0		
010	1	11	Mem[11010_2]
011	0		
100	0		
101	0		
110	1	10	Mem[10110_2]
111	0		

- **Lastly, 10010_2 is accessed**. The block indexed by **010** is valid, however, the tag bits of the word address, **10**, don't match the tag of the corresponding cache block, which is 11.

- This implies the block indexed by 010, in the cache, is storing the memory word at 11010_2 and not the memory word at 10010_2. Therefore, we have a cache **miss** and replace this block in the cache by a new block, i.e., the contents of 10010_2 in main memory. After updating the tag, the cache has been updated as follows.



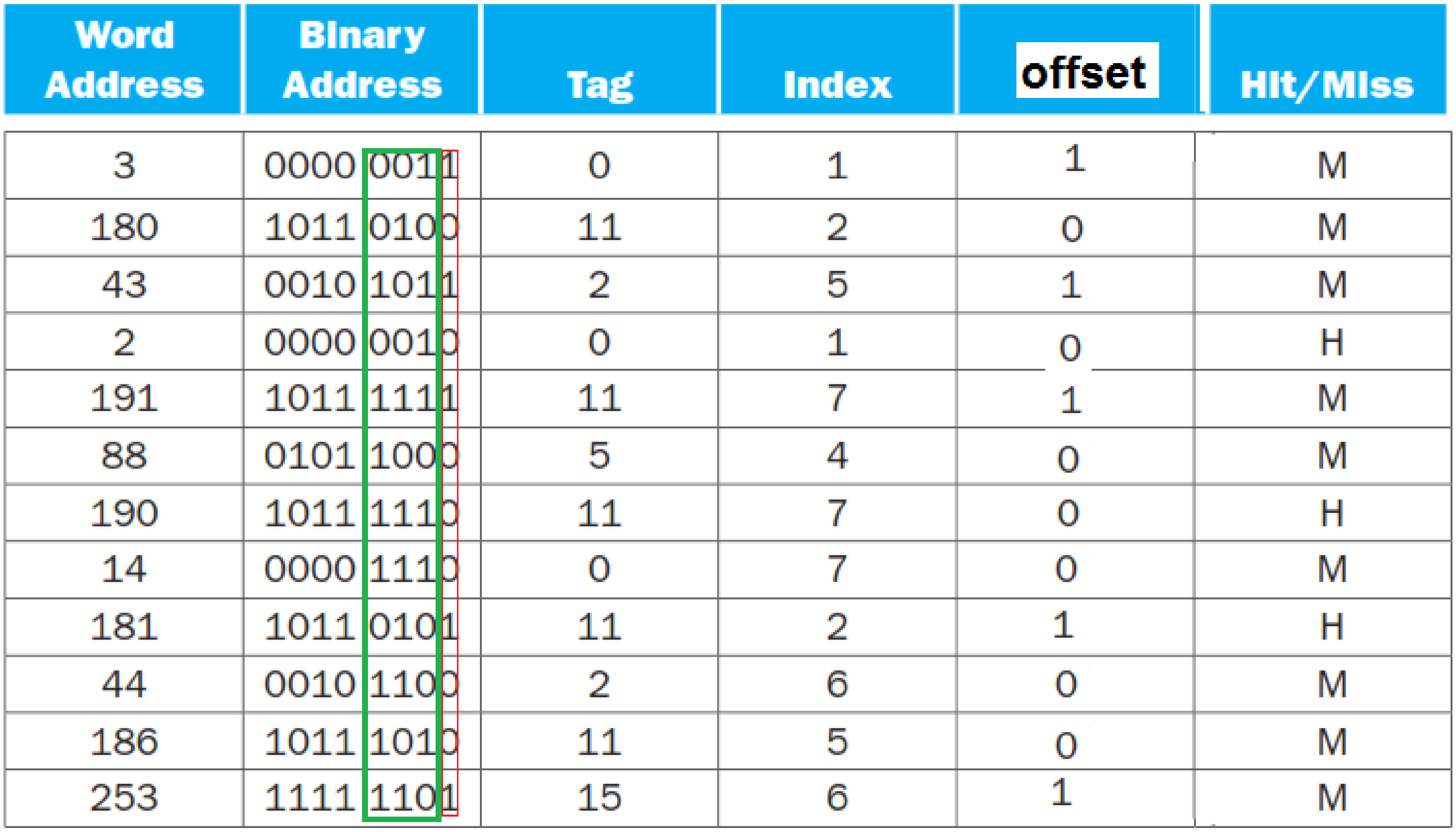
Index	V	Tag	Data (block = 32 bits)
000	1	10	Mem[10000_2]
001	0		
010	1	10	Mem[10010_2]
011	0		
100	0		
101	0		
110	1	10	Mem[10110_2]
111	0		



Exercise #	Below is a list of 32-bit memory address references, given as word addresses 3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253 For each of these references, identify the binary address, the tag, and the index given a direct-mapped cache with two-word blocks and a total size of 8 blocks. Also list if each reference is a hit or miss, assuming the cache is initially empty.
5.2.2	

Solution

The block size is 2 words, so you need 1 offset bit (because $2^1=2$). You have 8 blocks, so you need 3 index bits to give 8 different row indices (because $2^3=8$). That leaves you with the remaining 28 bits for the tag.



Word Address	Binary Address	Tag	Index	offset	Hit/Miss
3	0000 0011	0	1	1	M
180	1011 0100	11	2	0	M
43	0010 1011	2	5	1	M
2	0000 0010	0	1	0	H
191	1011 1111	11	7	1	M
88	0101 1000	5	4	0	M
190	1011 1110	11	7	0	H
14	0000 1110	0	7	0	M
181	1011 0101	11	2	1	H
44	0010 1100	2	6	0	M
186	1011 1010	11	5	0	M
253	1111 1101	15	6	1	M

Note: Shift right: 180 = 1011010$\underline{0}$ =1011010=90 the first bit: 0 (offset)

90 mod 8 =2

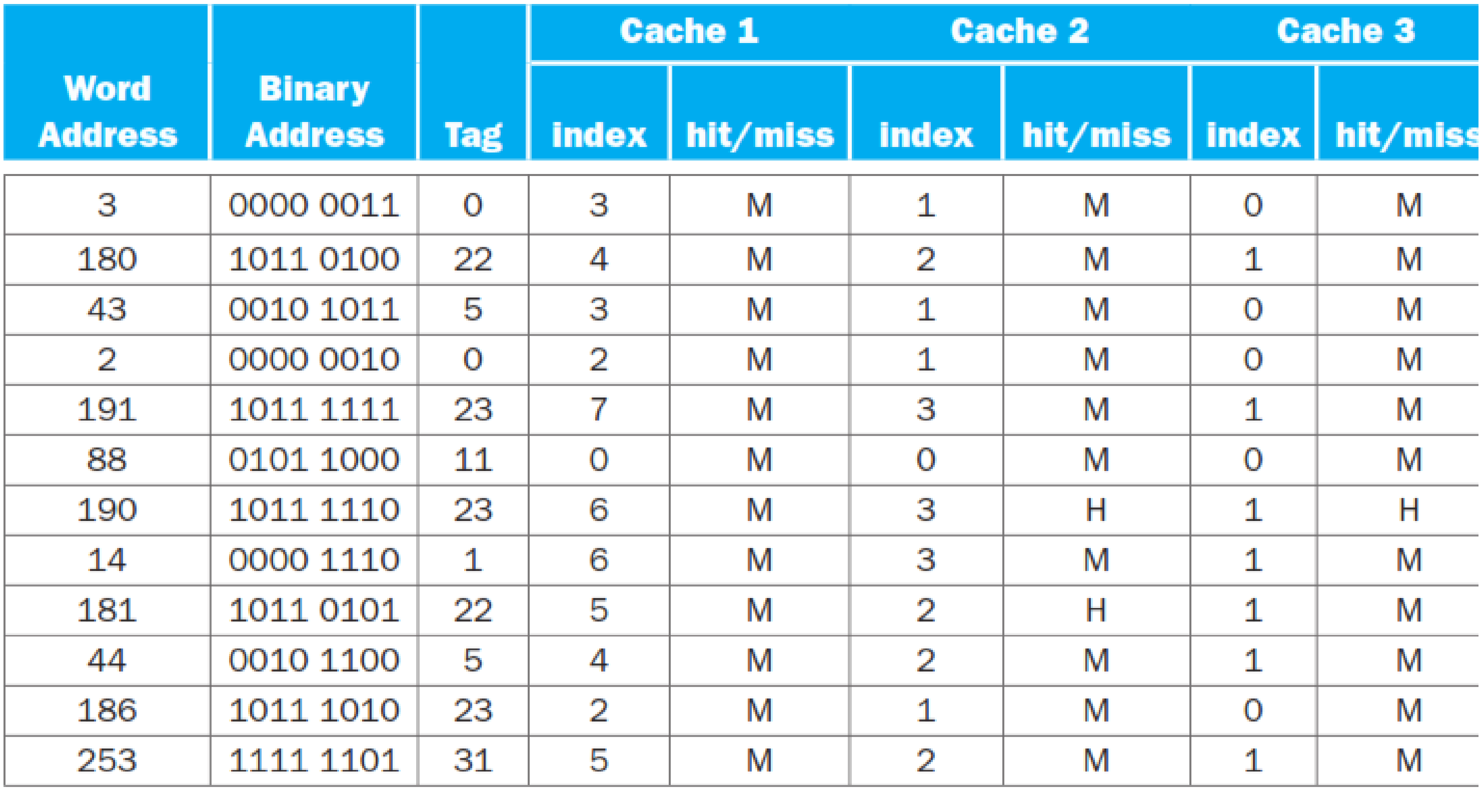
Shift right: 43 = 0010101$\underline{1}$ =0010101= 21 the first bit: 1 (offset)

21 mod 8 =5

Exercise 5.2.3

You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of 8 words of data: C1 has 1-word blocks, C2 has 2-word blocks, and C3 has 4-word blocks. In terms of miss rate, which cache design is the best? If the miss stall time is 25 cycles, and C1 has an access time of 2 cycles, C2 takes 3 cycles, and C3 takes 5 cycles, which is the best cache design?

Solution



Word Address	Binary Address	Tag	Cache 1		Cache 2		Cache 3	
			index	hit/miss	index	hit/miss	index	hit/miss
3	0000 0011	0	3	M	1	M	0	M
180	1011 0100	22	4	M	2	M	1	M
43	0010 1011	5	3	M	1	M	0	M
2	0000 0010	0	2	M	1	M	0	M
191	1011 1111	23	7	M	3	M	1	M
88	0101 1000	11	0	M	0	M	0	M
190	1011 1110	23	6	M	3	H	1	H
14	0000 1110	1	6	M	3	M	1	M
181	1011 0101	22	5	M	2	H	1	M
44	0010 1100	5	4	M	2	M	1	M
186	1011 1010	23	2	M	1	M	0	M
253	1111 1101	31	5	M	2	M	1	M

Cache 1 miss rate = 100% **(12 Word Address : miss)**

Cache 1 total cycles = 12 × 25 + 12 × 2 = 324

Cache 2 miss rate = 10/12 = 83%

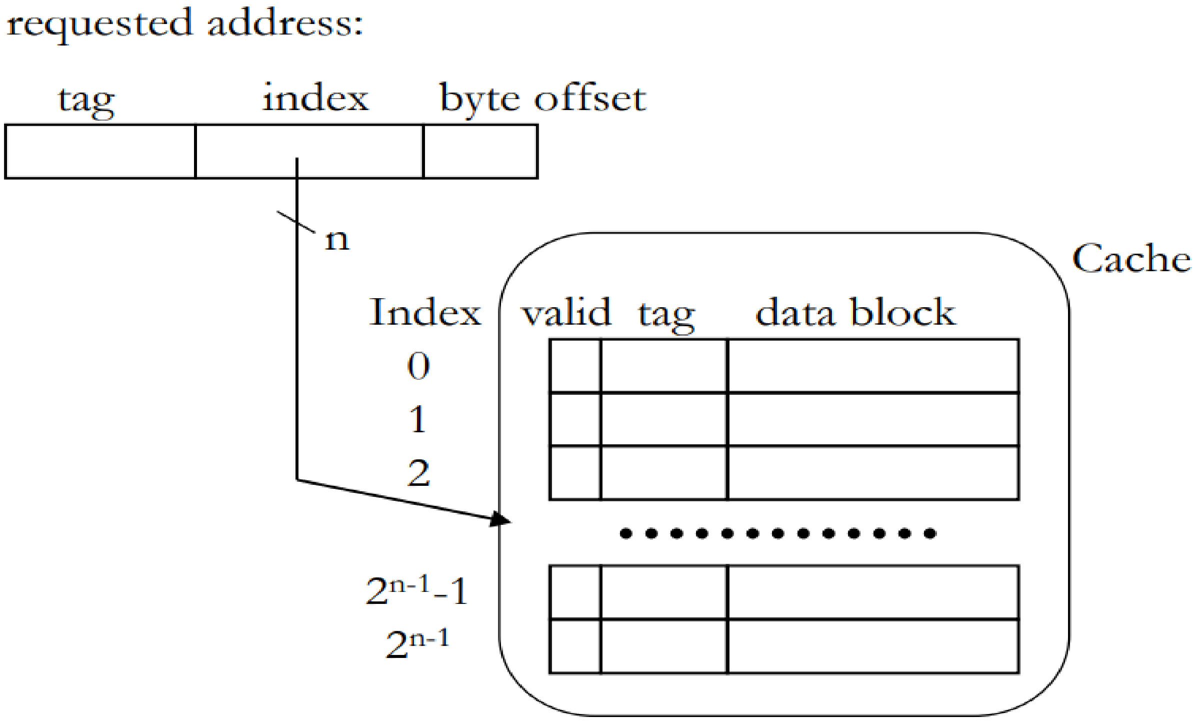
Cache 2 total cycles = 10 × 25 + 12 × 3 = 286

Cache 3 miss rate = 11/12 = 92%

Cache 3 total cycles = 11 × 25 + 12 × 5 = 335

Cache 2 provides the best performance.

Address mapping for direct-mapped cache



Bits in a Cache

The total number of bits needed for a cache is a function of the **cache size** and the **address size**, because the cache includes both the storage for the **data** and the **tags**. For the following situation:

- 32-bit addresses
- A direct-mapped cache
- The cache size is 2^n blocks, so *n* bits are used for the index
- The block size is 2^m words or 2^{m+2} bytes, so *m* bits are used for the word

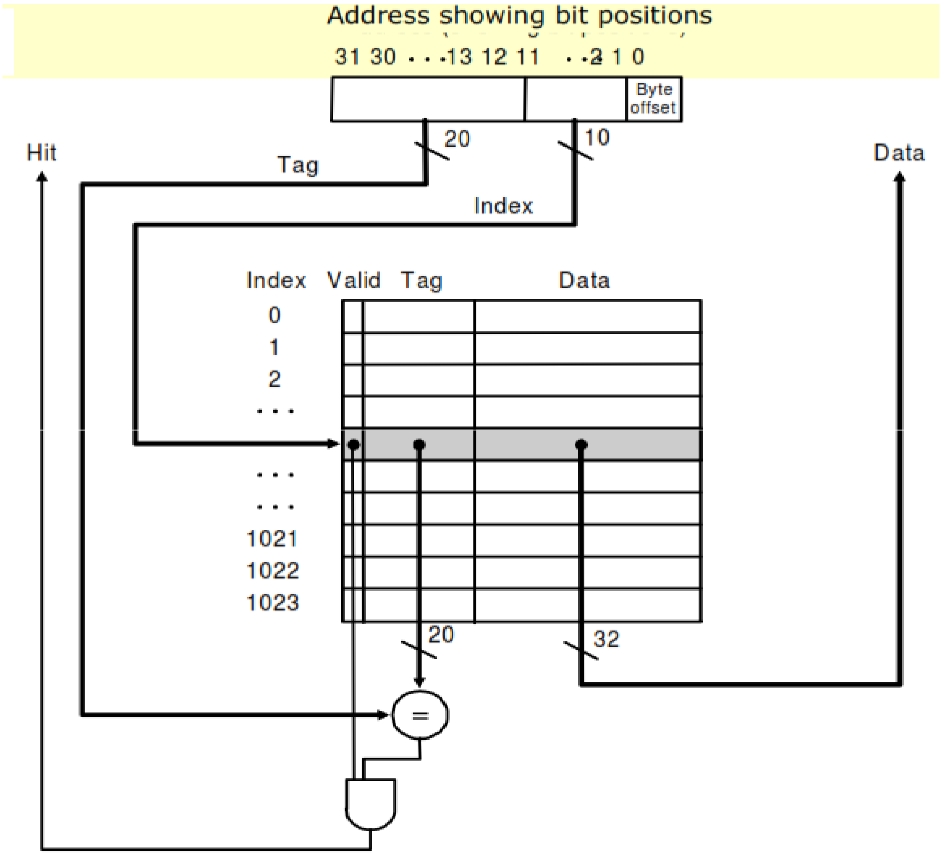
within the block, and two bits are used for the byte part of the address

- The size of the tag field is : **32 - (n +*m* + 2)**
- The total number of bits in a direct-mapped cache is

2^n * (block size + tag size + valid field size)

❑ Example:

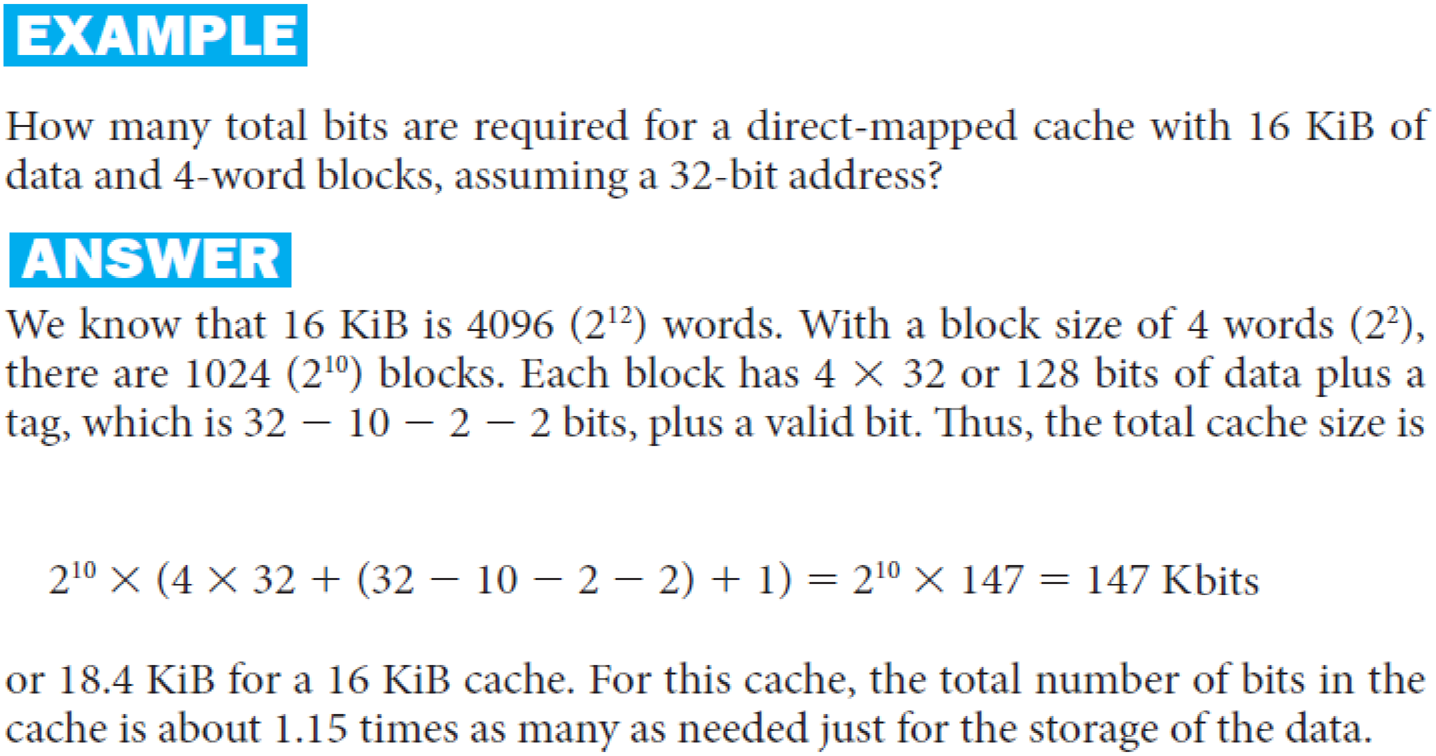
Cache with 1024 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 10 bits used to index into cache



- ➢ This cache holds 1024 words or 4 KB, because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache.

- ➢ We assume 32-bit addresses in this example. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address, leaving $32 - 10 - 2 = 20$ bits to be compared against the tag.

- ➢ If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.



EXAMPLE

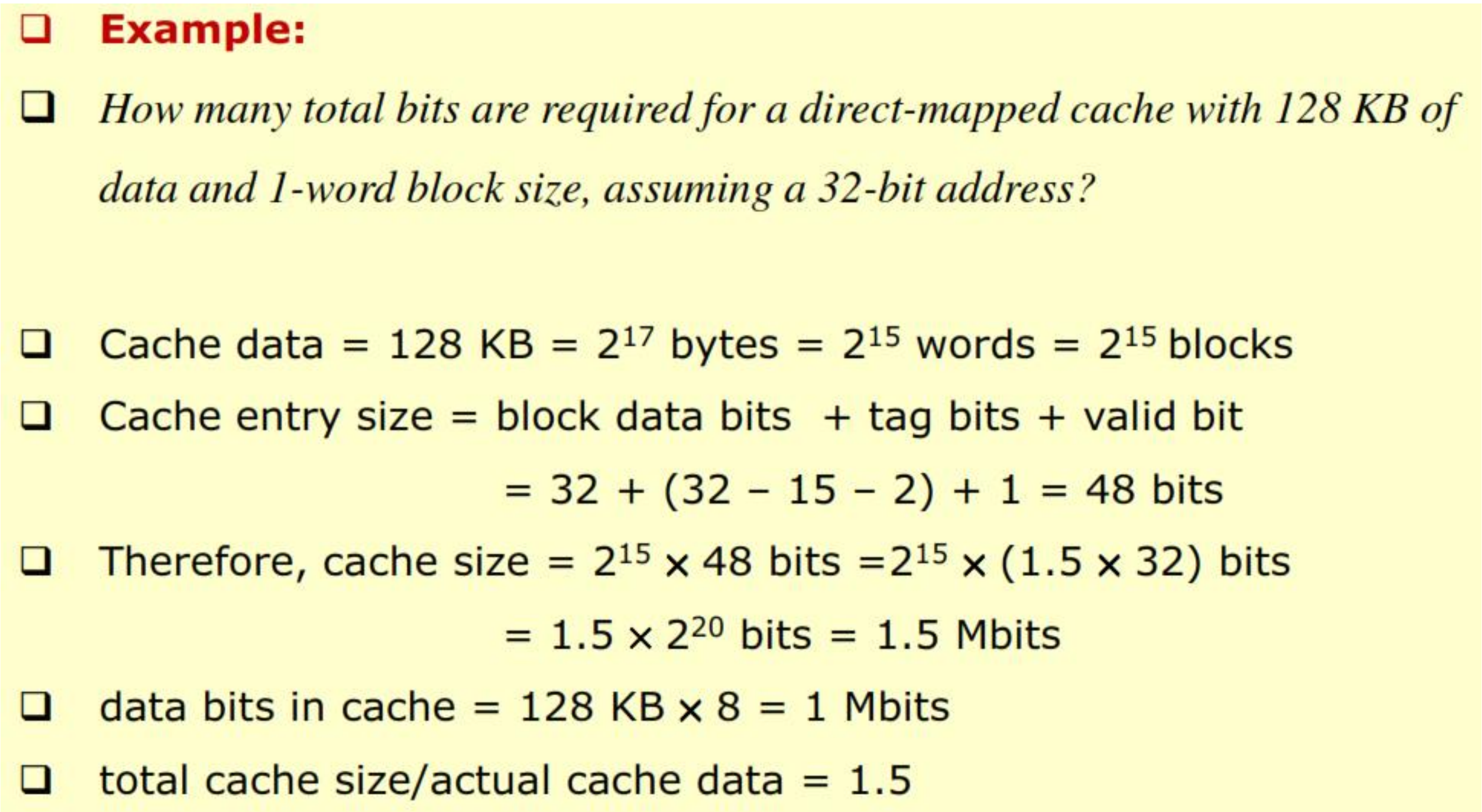
How many total bits are required for a direct-mapped cache with 16 KiB of data and 4-word blocks, assuming a 32-bit address?

ANSWER

We know that 16 KiB is 4096 (2^{12}) words. With a block size of 4 words (2^2), there are 1024 (2^{10}) blocks. Each block has 4×32 or 128 bits of data plus a tag, which is $32 - 10 - 2 - 2$ bits, plus a valid bit. Thus, the total cache size is

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kbits}$$

or 18.4 KiB for a 16 KiB cache. For this cache, the total number of bits in the cache is about 1.15 times as many as needed just for the storage of the data.



❑ **Example:**

❑ *How many total bits are required for a direct-mapped cache with 128 KB of data and 1-word block size, assuming a 32-bit address?*

❑ Cache data = 128 KB = 2^{17} bytes = 2^{15} words = 2^{15} blocks

❑ Cache entry size = block data bits + tag bits + valid bit

= 32 + (32 – 15 – 2) + 1 = 48 bits

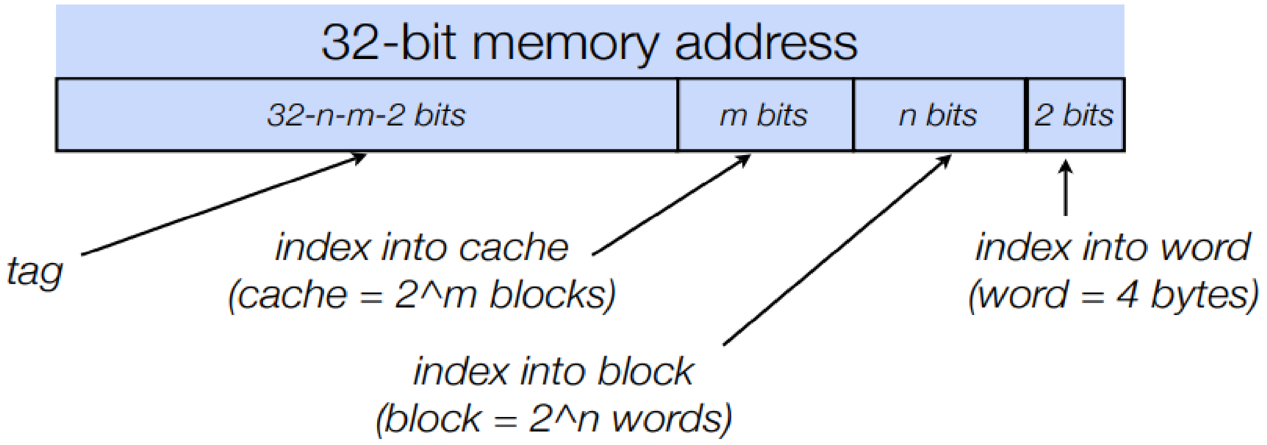
❑ Therefore, cache size = $2^{15} \times 48$ bits $= 2^{15} \times (1.5 \times 32)$ bits

$= 1.5 \times 2^{20}$ bits = 1.5 Mbits

❑ data bits in cache = 128 KB x 8 = 1 Mbits

❑ total cache size/actual cache data = 1.5

Multi-word Cache Blocks (Direct Mapping)



32-bit memory address

32-n-m-2 bits | m bits | n bits | 2 bits

tag

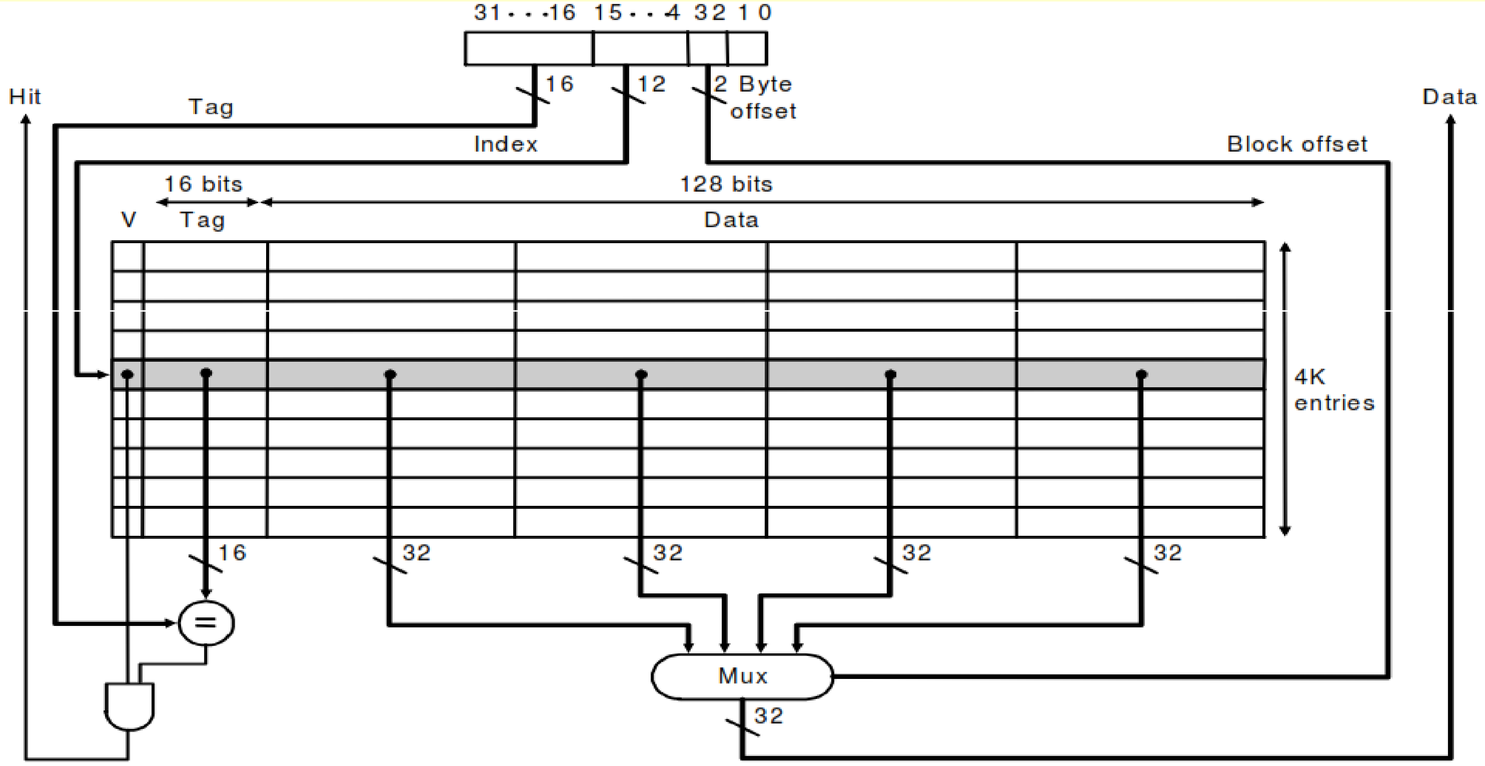
index into cache (cache = 2^m blocks)

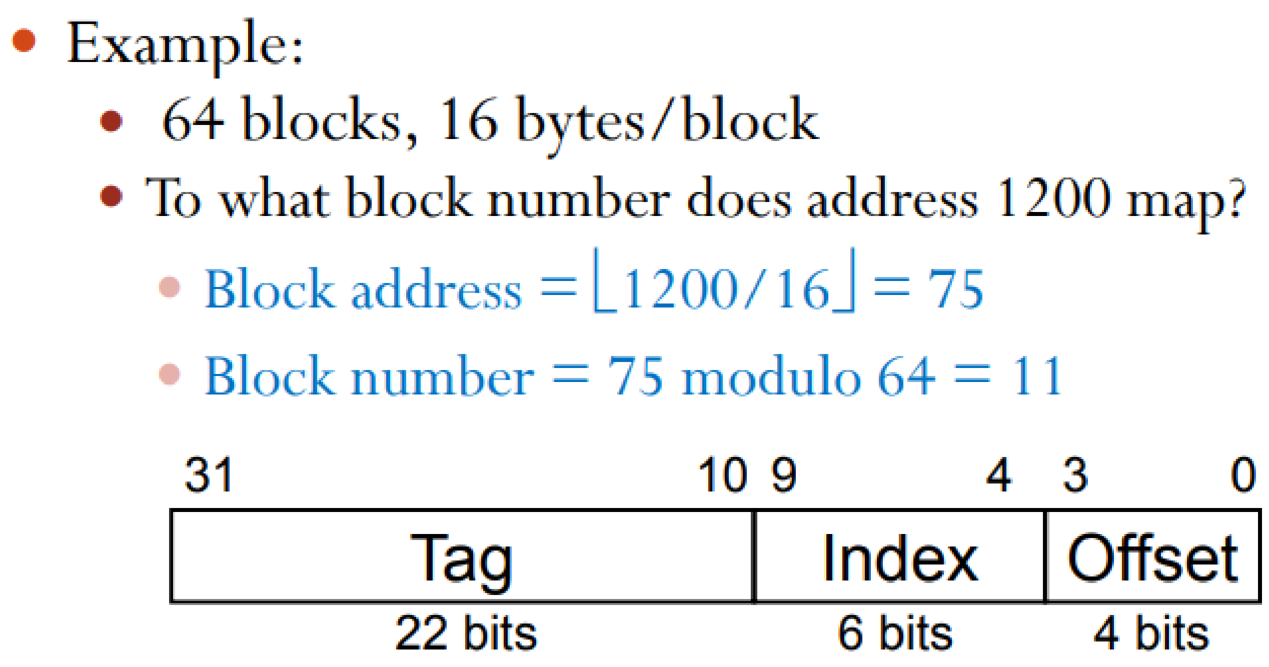
index into block (block = 2^n words)

index into word (word = 4 bytes)

- e.g., m=5, n=4 (16 words per block, 32 blocks in cache: cache stores 32*16 words)
  - 110110001010100110101 11001 1010 01:
    - byte #1 of 10th word in 25th block
  - All words whose address is prefixed with 110110001010100110101 11001 moved into the 25th block of the cache simultaneously

❑ Cache with 4K 4-word blocks: first 2 bits are *byte offset* is ignored, next 2 bits are *block offset*, and the next 12 bits are used to index into cache



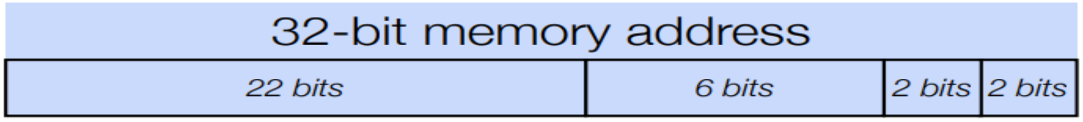


- Example:
  - 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
    - Block address = $\lfloor 1200/16 \rfloor = 75$
    - Block number = 75 modulo 64 = 11

31 … 10	9 … 4	3 … 0
Tag	Index	Offset
22 bits	6 bits	4 bits

This block maps all addresses between 1200 and 1215

- Block address = floor(1200/16) = 75 (75th block in memory)
- Block number = 75 modulo 64 = 11 (Direct mapping, would map to 11th block in cache)



32-bit memory address			
22 bits	*6 bits*	*2 bits*	*2 bits*

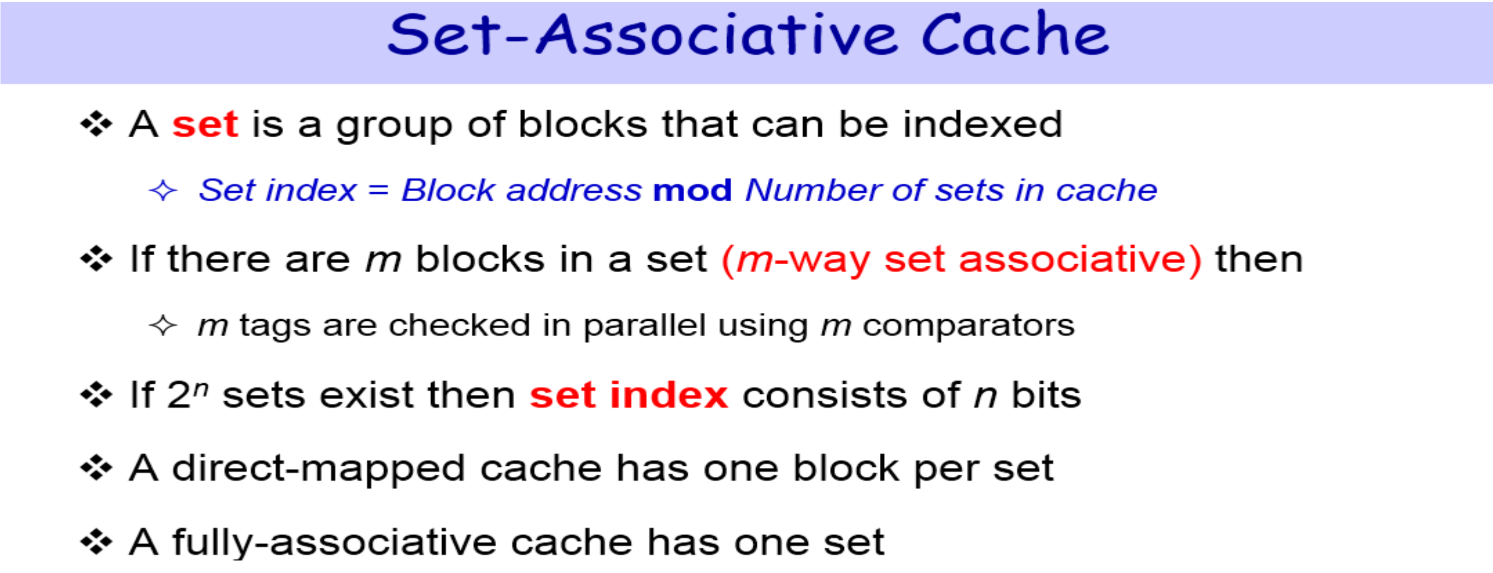
We first find out the memory block number that byte address 1200 belongs to. Since the size of a block is 16 bytes.

Byte address 0 to 15:	block 0
Byte address 16 to 31:	block 1
Byte address 32 to 47:	block 2, and so on.

Byte address 1200 will belong to block number: floor(1200/16) = 75.

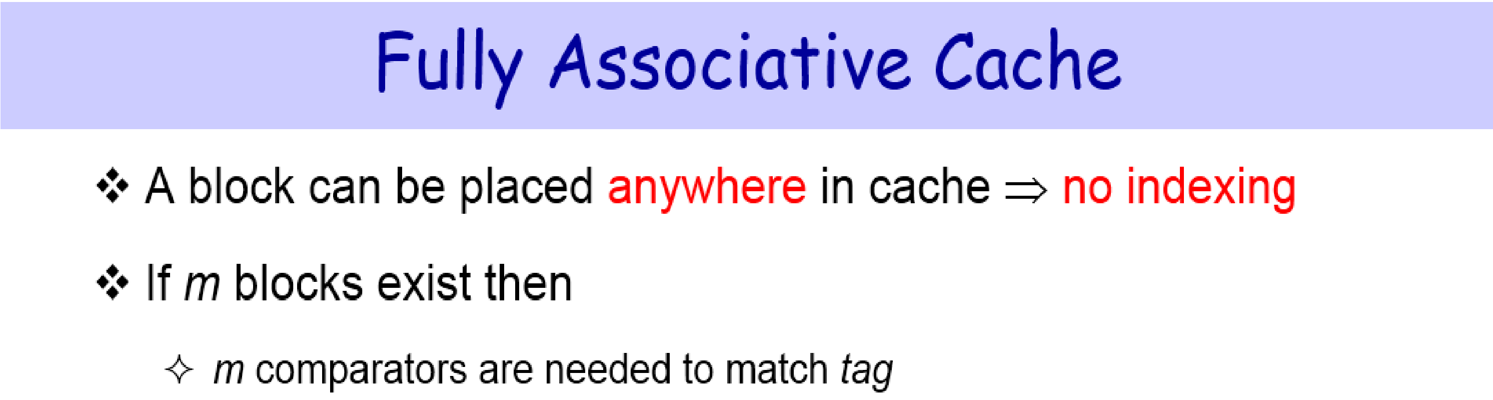
For direct mapped cache,

Cache block no. = (Memory block no.) MOD (No. of cache blocks)
= 75 MOD 64 = 11.



Set-Associative Cache

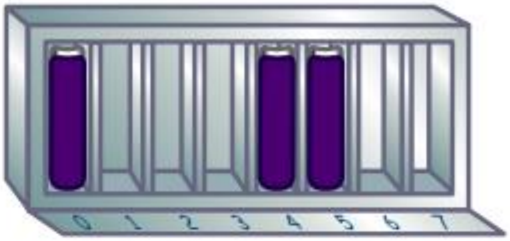
- A **set** is a group of blocks that can be indexed
  - *Set index = Block address* **mod** *Number of sets in cache*
- If there are *m* blocks in a set (*m*-way set associative) then
  - *m* tags are checked in parallel using *m* comparators
- If 2^n sets exist then **set index** consists of *n* bits
- A direct-mapped cache has one block per set
- A fully-associative cache has one set



Fully Associative Cache

- A block can be placed anywhere in cache ⇒ no indexing
- If *m* blocks exist then
  - *m* comparators are needed to match *tag*

Direct Mapped

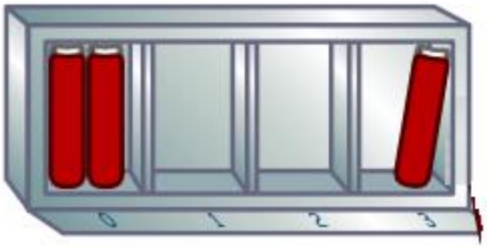


Memory Address

Tag	Index	Offset

A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

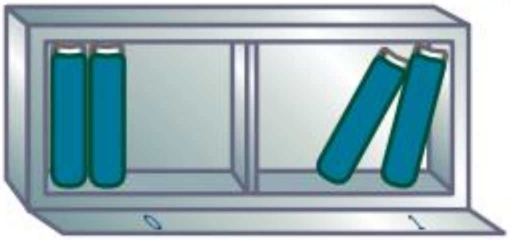
2-Way Set Associative



Tag	Index	Offset

This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

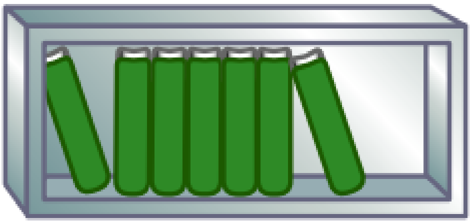
4-Way Set Associative



Tag	Index	Offset

Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.

Fully Associative

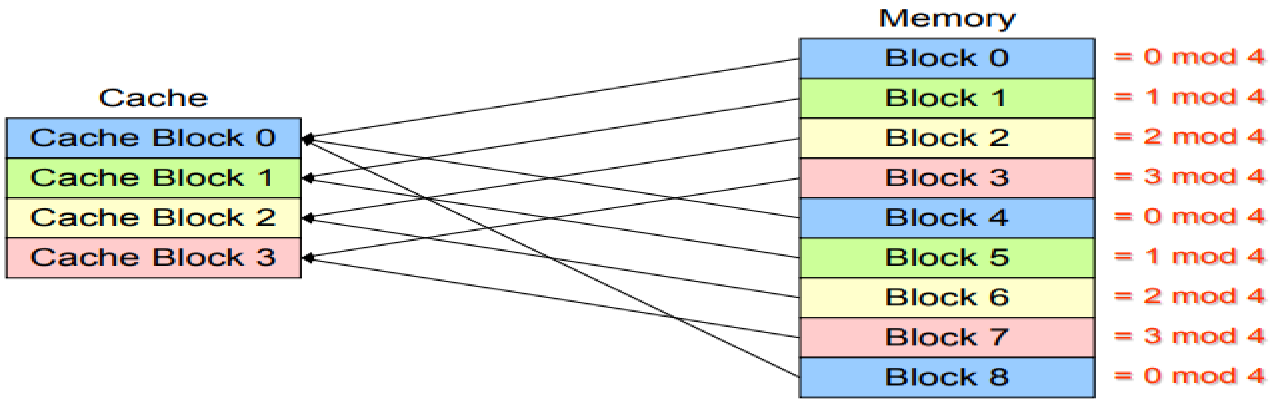


Tag	Offset

No index is needed, since a cache block can go anywhere in the cache. Every tag must be compared when finding a block in the cache, but block placement is very flexible!

Direct Mapping

- Each block from memory can only be put in one location
- Given n cache blocks,
  MM block i maps to cache block i mod n



Memory

Cache

Cache Block 0

Cache Block 1

Cache Block 2

Cache Block 3

Block 0 = 0 mod 4

Block 1 = 1 mod 4

Block 2 = 2 mod 4

Block 3 = 3 mod 4

Block 4 = 0 mod 4

Block 5 = 1 mod 4

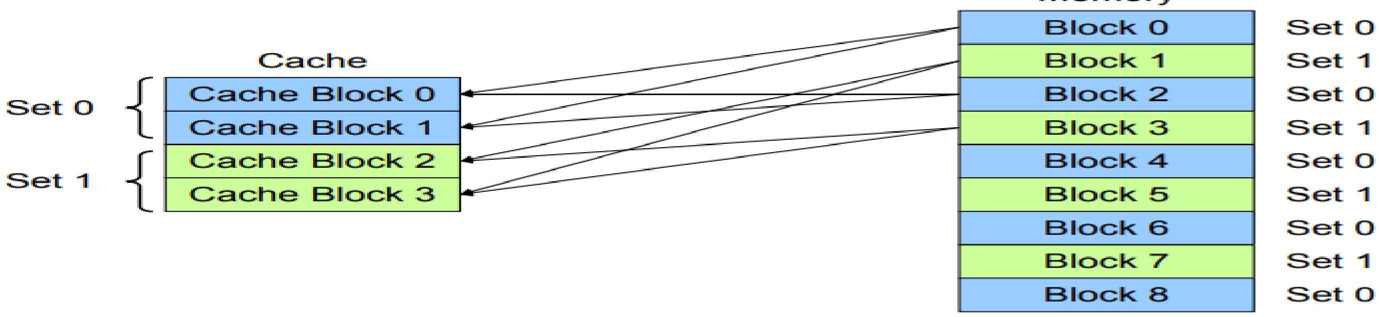
Block 6 = 2 mod 4

Block 7 = 3 mod 4

Block 8 = 0 mod 4

K-way Set-Associative Mapping

- Given, S sets, block i of MM maps to set i mod s
- Within the set, block can be put anywhere
- Let k = number of cache blocks per set = n/s
  - K comparisons required for search



Memory

Cache

Set 0 Cache Block 0

Cache Block 1

Set 1 Cache Block 2

Cache Block 3

Block 0 Set 0

Block 1 Set 1

Block 2 Set 0

Block 3 Set 1

Block 4 Set 0

Block 5 Set 1

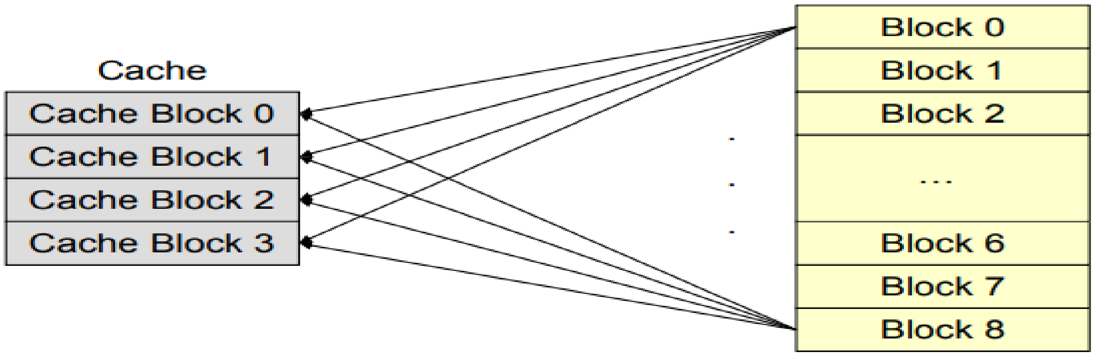
Block 6 Set 0

Block 7 Set 1

Block 8 Set 0

Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no restriction)
  - Implies we have to search everywhere to determine hit or miss



Memory

Cache

Cache Block 0

Cache Block 1

Cache Block 2

Cache Block 3

Block 0

Block 1

Block 2

. . .

Block 6

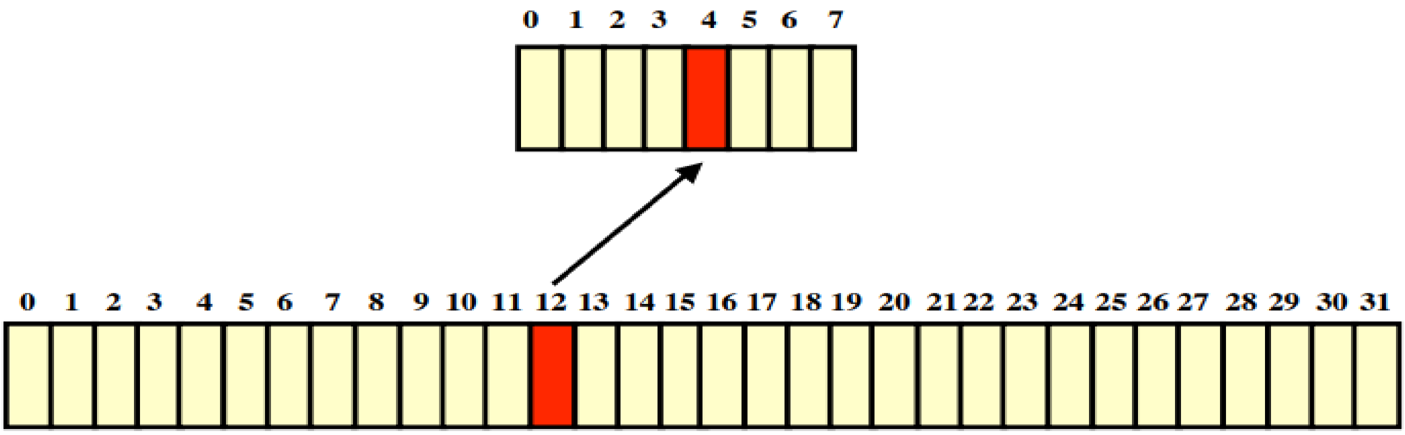
Block 7

Block 8

Direct Mapped

- **Each block mapped to exactly 1 cache location**

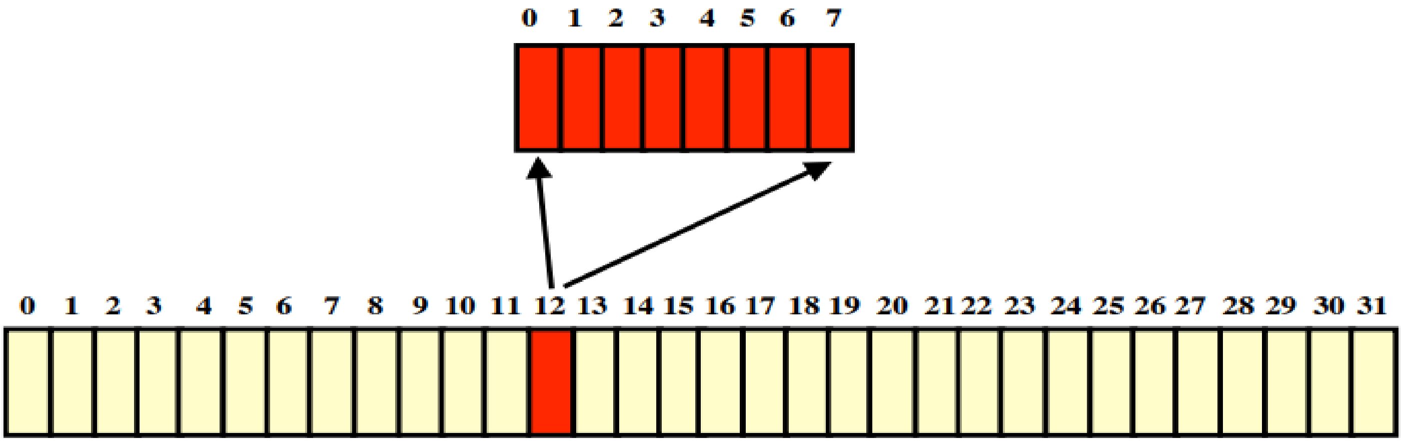
Cache location = (block address) MOD (# blocks in cache)



Fully Associative

- **Each block mapped to any cache location**

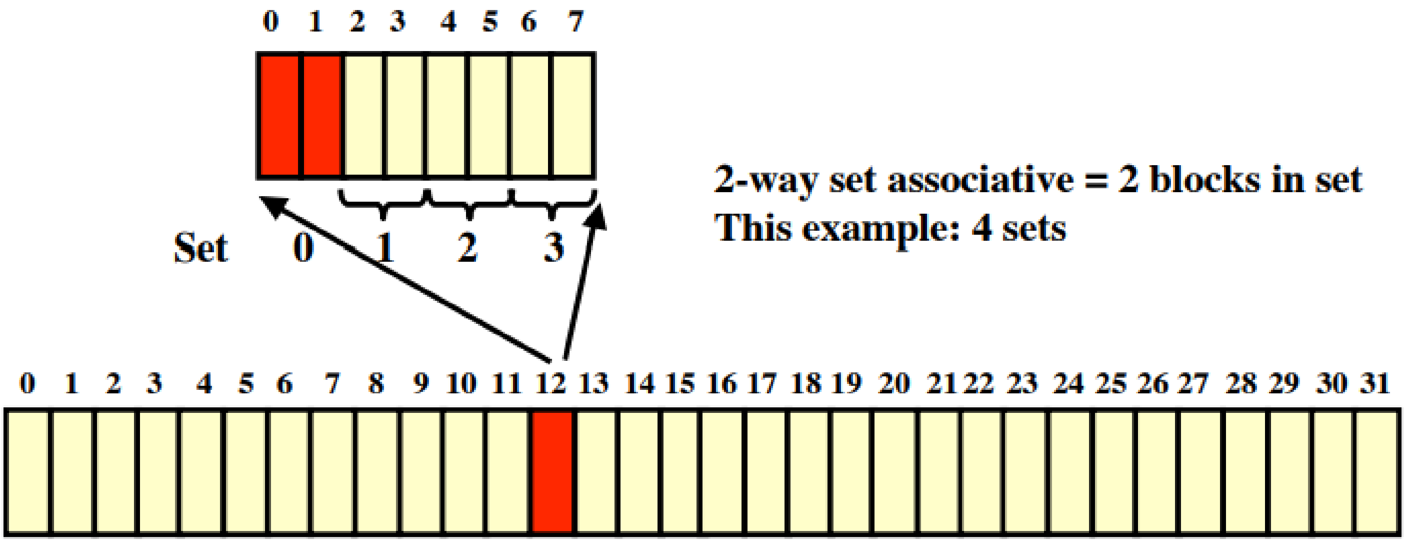
Cache location = any



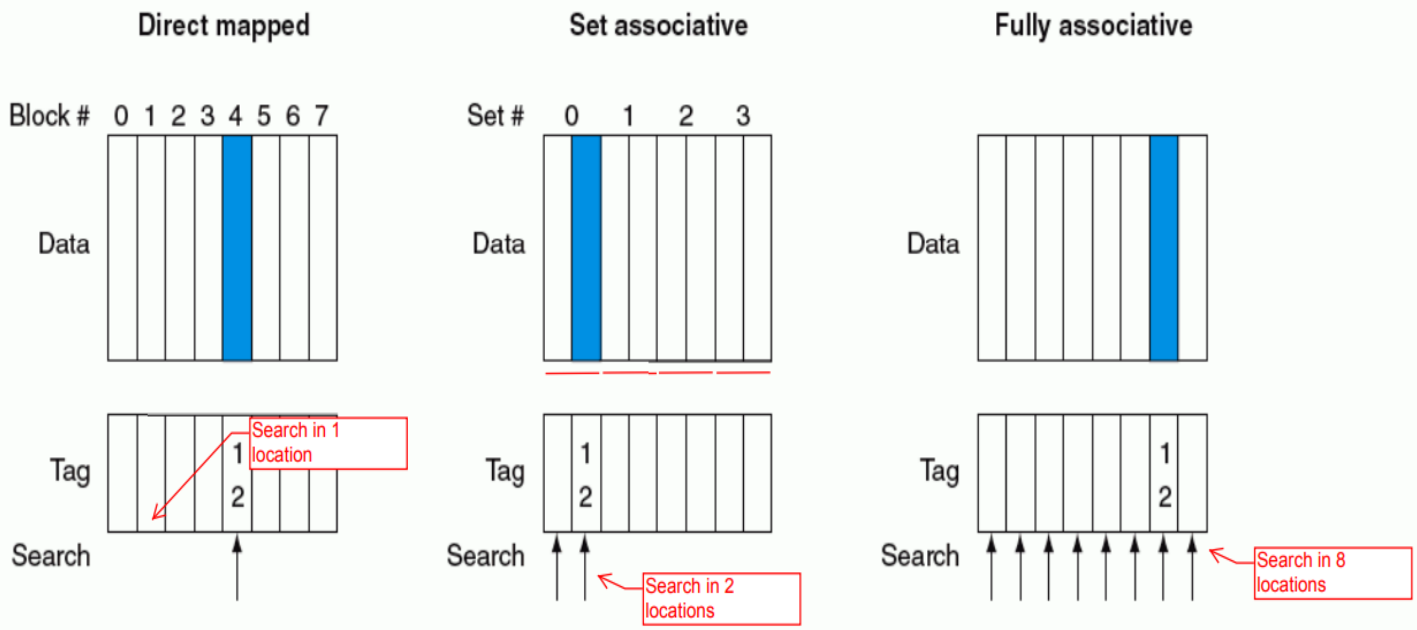
Set Associative

- **Each block mapped to subset of cache locations**

Set selection = (block address) MOD (# sets in cache)



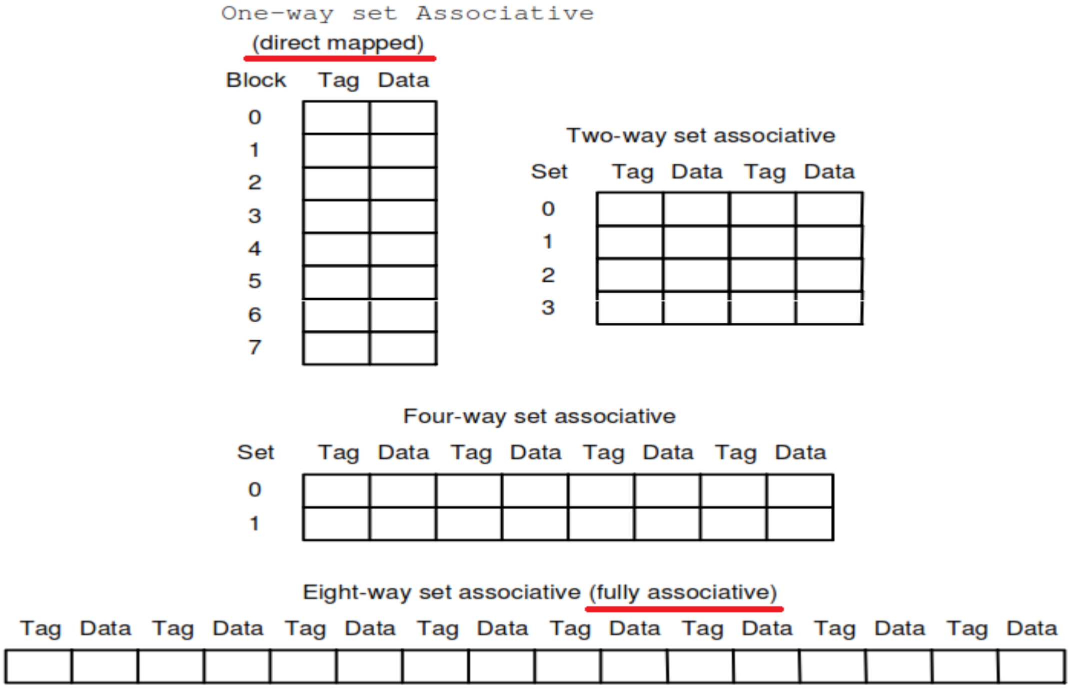
2-way set associative = 2 blocks in set
This example: 4 sets



The location of a memory block whose address is 12 in a cache with 8 blocks varies for direct-mapped, set associative , and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

Spectrum of Associativity

❑ For a cache with 8 entries (8-block) with different degrees of associativity:


One-way set Associative
(direct mapped)
Block Tag Data
0
1
2
3
4
5
6
7
Two-way set associative
Set Tag Data Tag Data
0
1
2
3
Four-way set associative
Set Tag Data Tag Data Tag Data Tag Data
0
1
Eight-way set associative (fully associative)
Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data

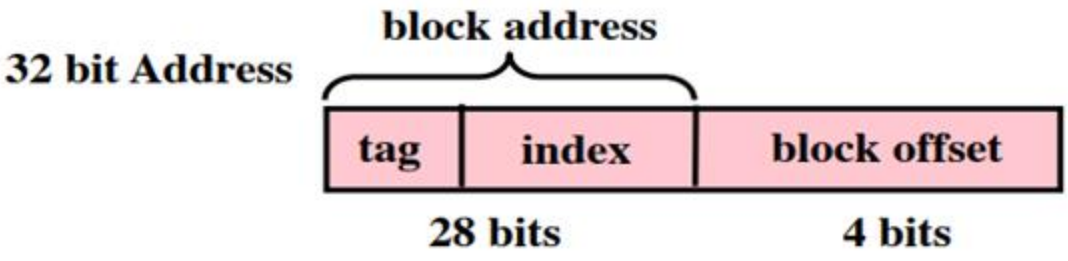
	# of sets	Blocks per set
Direct mapped	# of blocks in cache	1
Set associative	(# of blocks in cache)/ associativity	Associativity (typically 2 to 16)
Fully associative	1	# of blocks in cache

	Location method	# of comparisons
Direct mapped	Index	1
Set associative	Index the set; compare set's tags	Degree of associativity
Fully associative	Compare all blocks tags	# of blocks

How Do We Find a Block in The Cache?

- Our Example:
  - Main memory address space = 32 bits (= 4GBytes)
  - Block size = 4 words = 16 bytes
  - Cache capacity = 8 blocks = 128 bytes

-



32 bit Address

block address

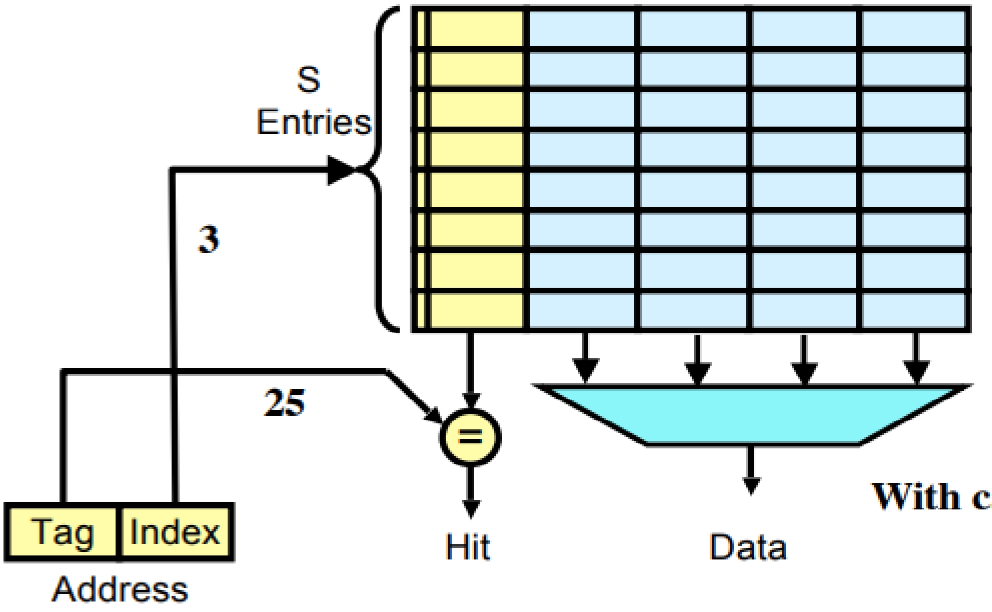
tag | index | block offset

28 bits

4 bits

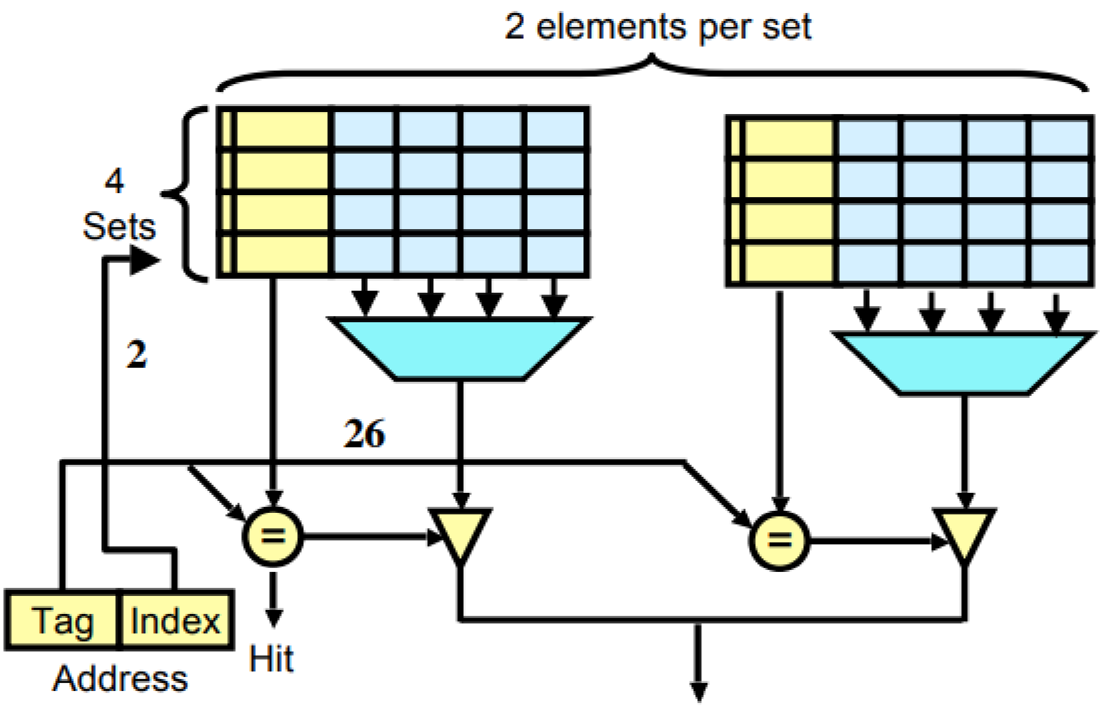
- index ⇒ which set
- tag ⇒ which data/instruction in block
- block offset ⇒ which word in block

Finding a Block: Direct-Mapped

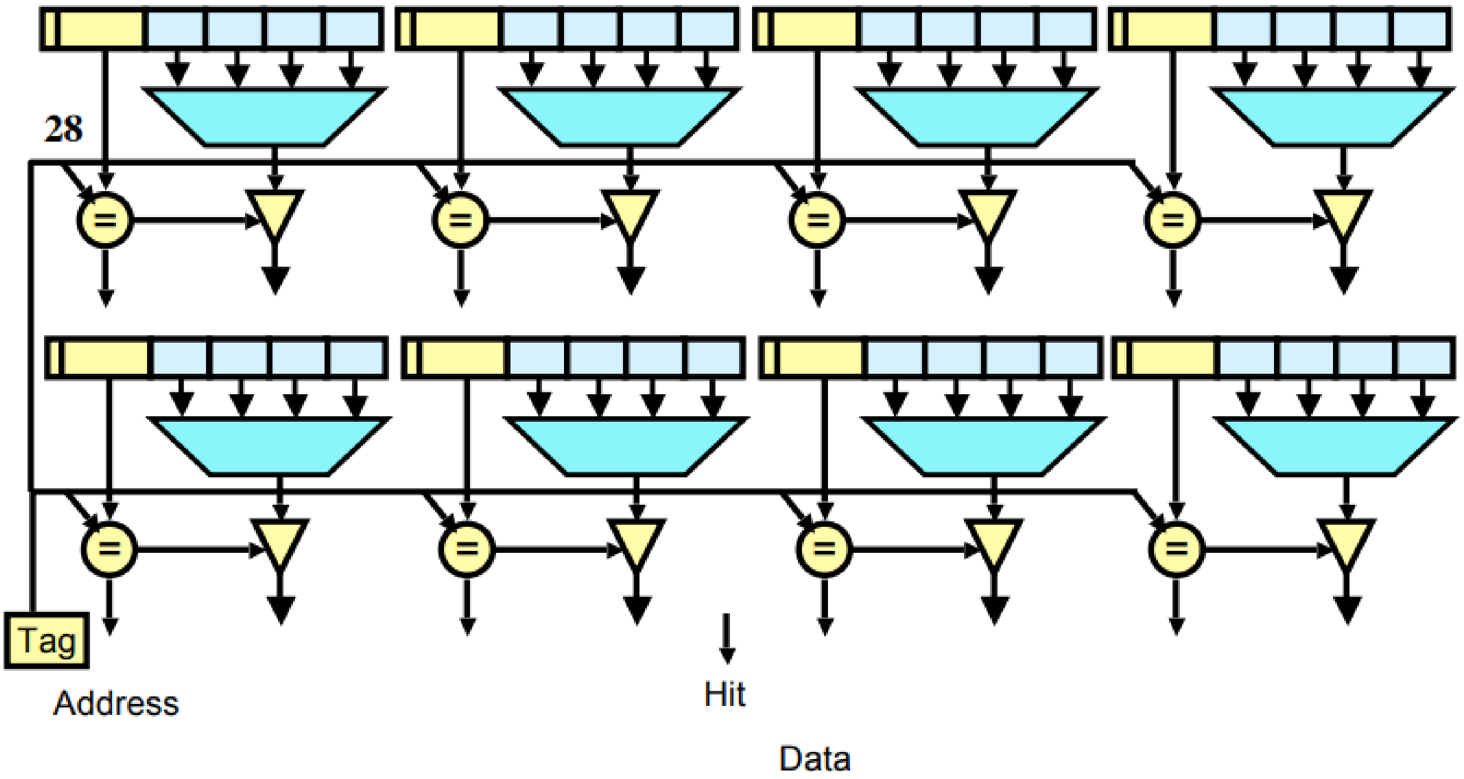


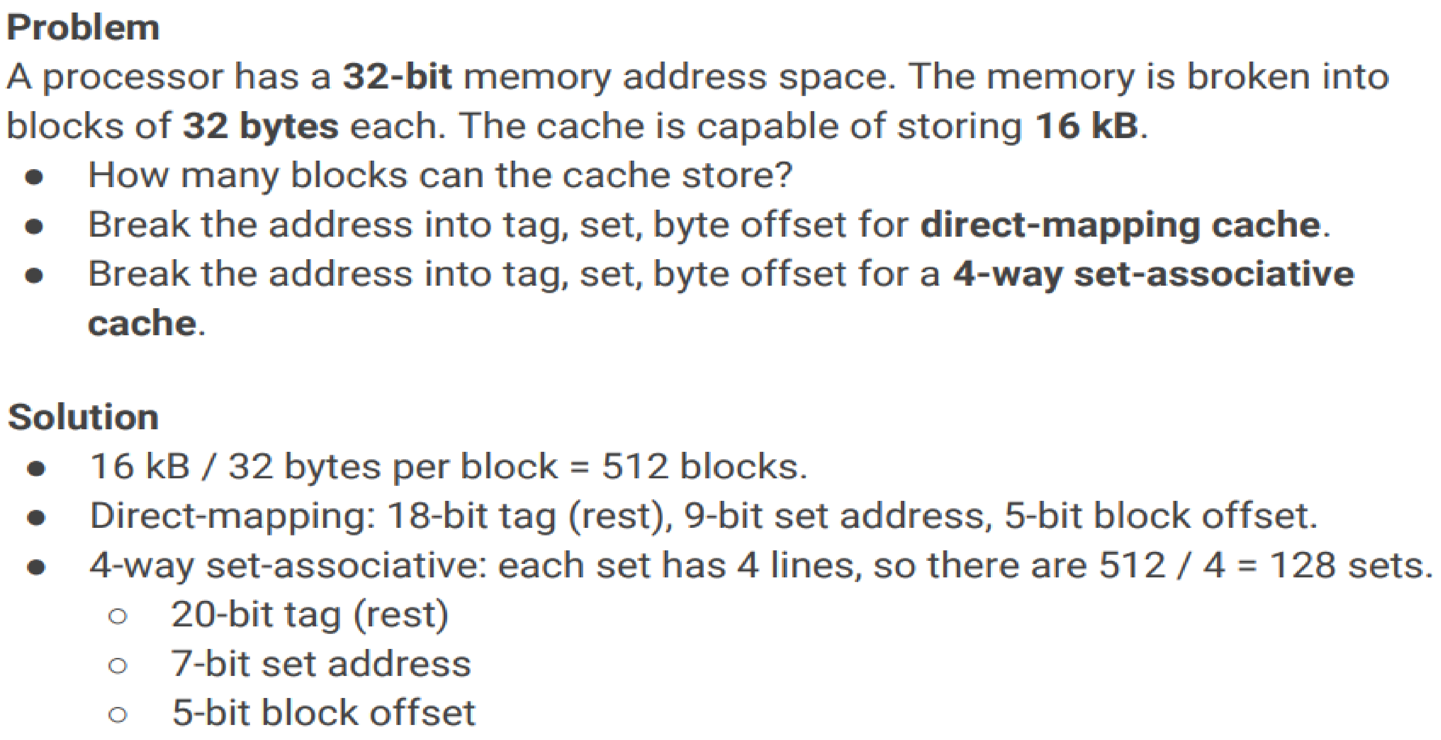
With cache capacity = 8 blocks

Finding A Block: 2-Way Set-Associative



Finding A Block: Fully Associative





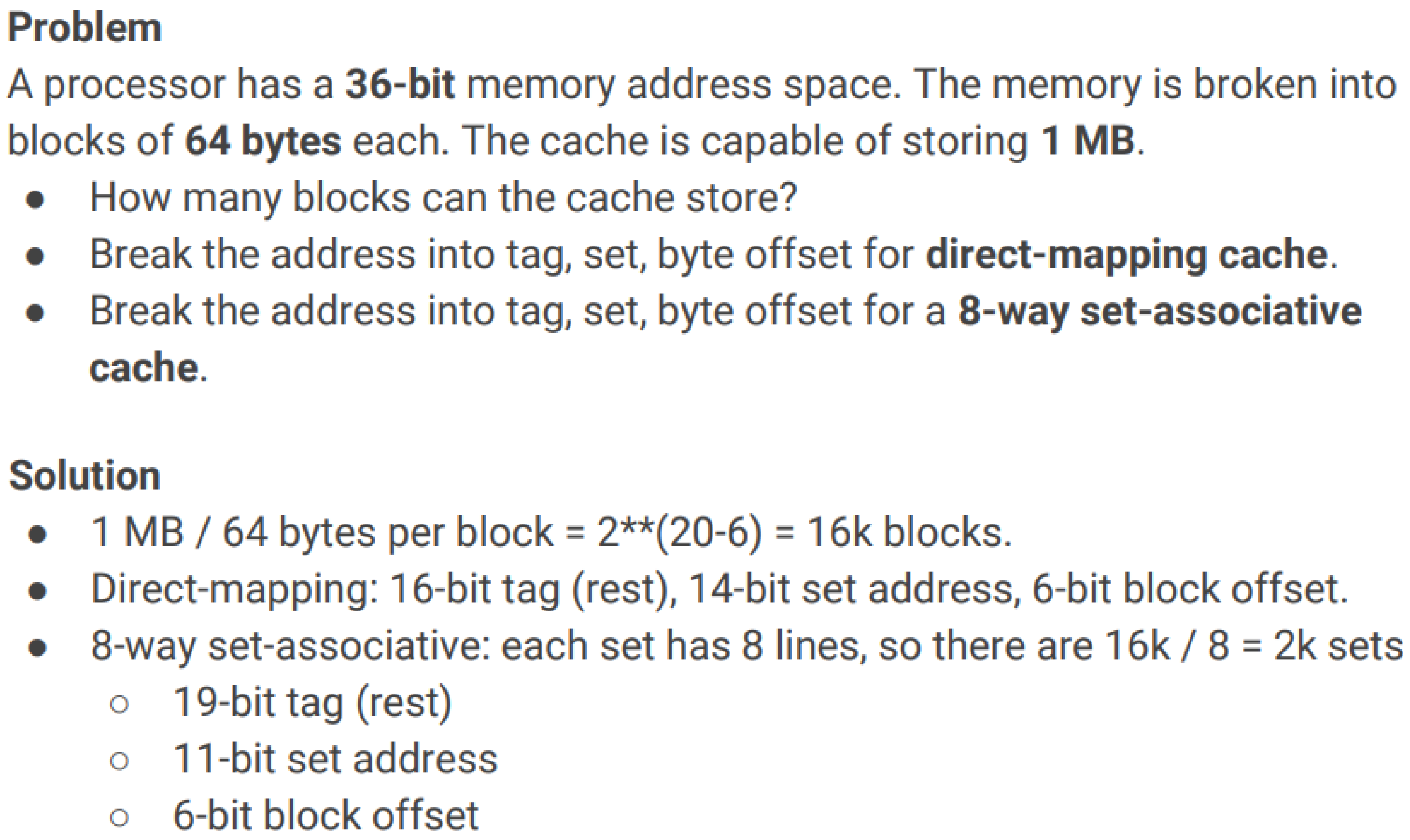
Problem

A processor has a **32-bit** memory address space. The memory is broken into blocks of **32 bytes** each. The cache is capable of storing **16 kB**.

- How many blocks can the cache store?
- Break the address into tag, set, byte offset for **direct-mapping cache**.
- Break the address into tag, set, byte offset for a **4-way set-associative cache**.

Solution

- 16 kB / 32 bytes per block = 512 blocks.
- Direct-mapping: 18-bit tag (rest), 9-bit set address, 5-bit block offset.
- 4-way set-associative: each set has 4 lines, so there are 512 / 4 = 128 sets.
  - 20-bit tag (rest)
  - 7-bit set address
  - 5-bit block offset



Problem

A processor has a **36-bit** memory address space. The memory is broken into blocks of **64 bytes** each. The cache is capable of storing **1 MB**.

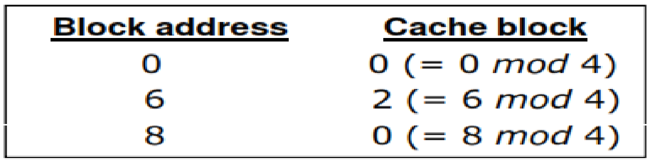
- How many blocks can the cache store?
- Break the address into tag, set, byte offset for **direct-mapping cache**.
- Break the address into tag, set, byte offset for a **8-way set-associative cache**.

Solution

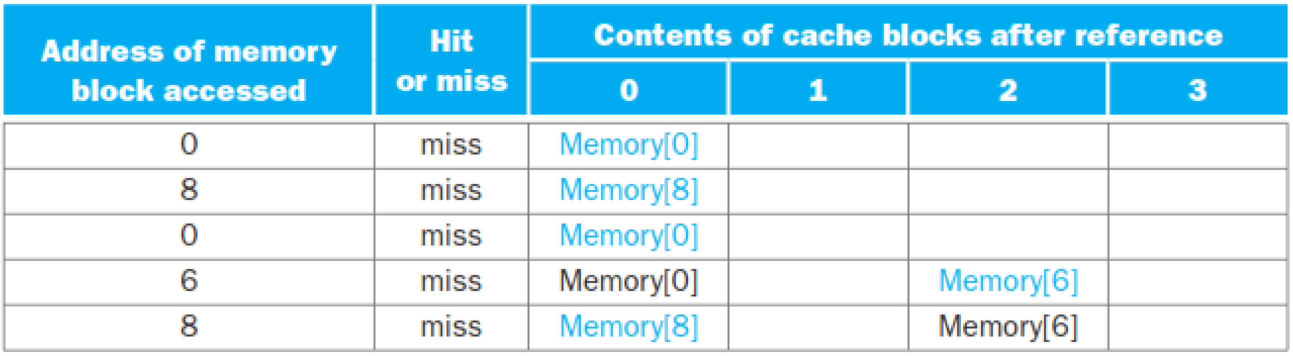
- 1 MB / 64 bytes per block = 2**(20-6) = 16k blocks.
- Direct-mapping: 16-bit tag (rest), 14-bit set address, 6-bit block offset.
- 8-way set-associative: each set has 8 lines, so there are 16k / 8 = 2k sets
  - 19-bit tag (rest)
  - 11-bit set address
  - 6-bit block offset

Example

- ❑ Compare 4-block caches
  - ❑ Direct mapped, 2-way set associative, fully associative
  - ❑ Block access sequence: 0, 8, 0, 6, 8
- ❑ **Direct mapped:**

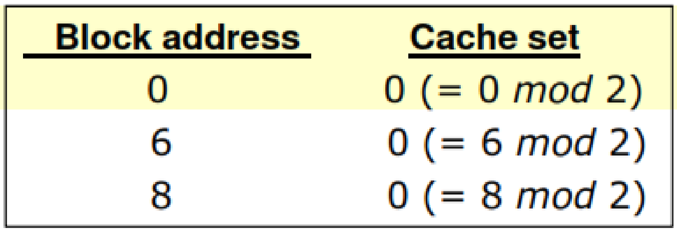


Block address	Cache block
0	0 (= 0 *mod* 4)
6	2 (= 6 *mod* 4)
8	0 (= 8 *mod* 4)

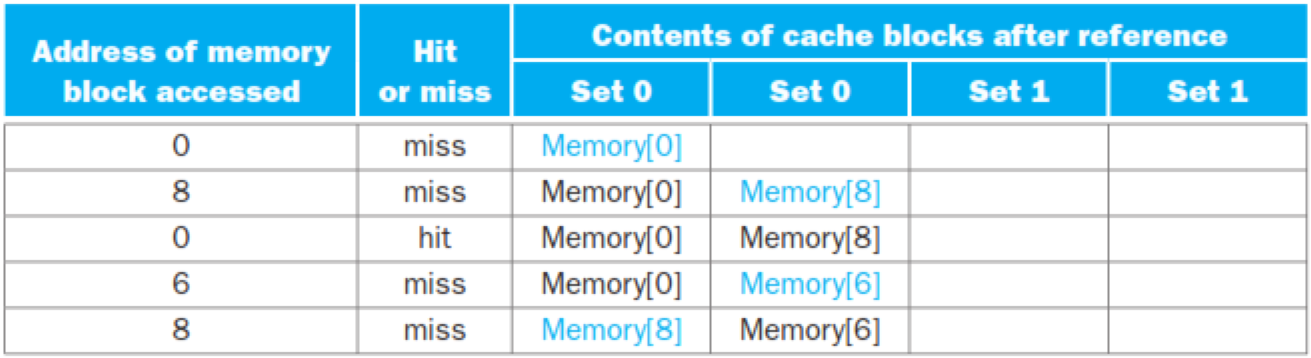


Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

- ❑ 2-way set associative:



Block address	Cache set
0	0 (= 0 *mod* 2)
6	0 (= 6 *mod* 2)
8	0 (= 8 *mod* 2)



Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Choosing Which Block to Replace

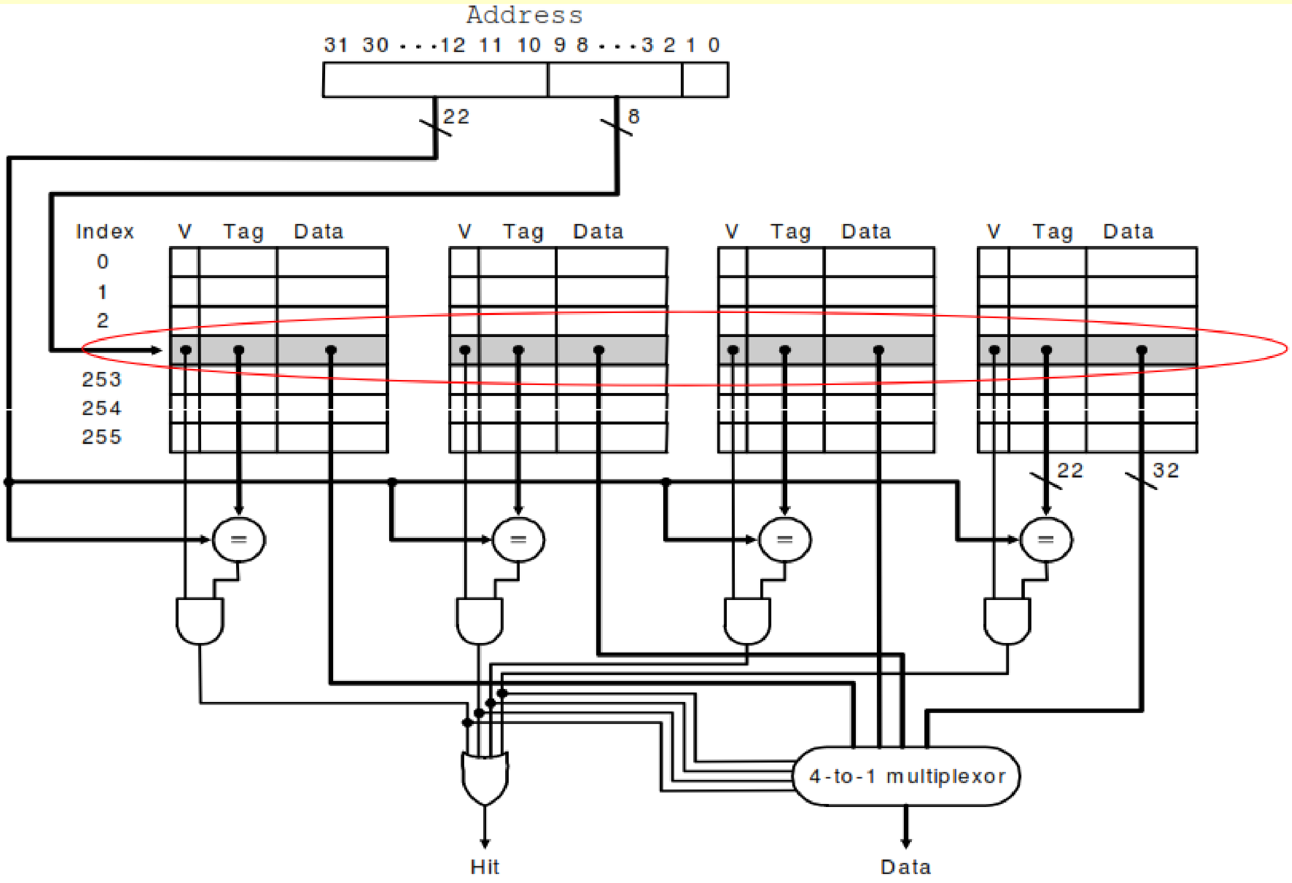
Least Recently Used (LRU) A replacement scheme in which the block replaced is the one that has been unused for the longest time.

❑ Fully associative:

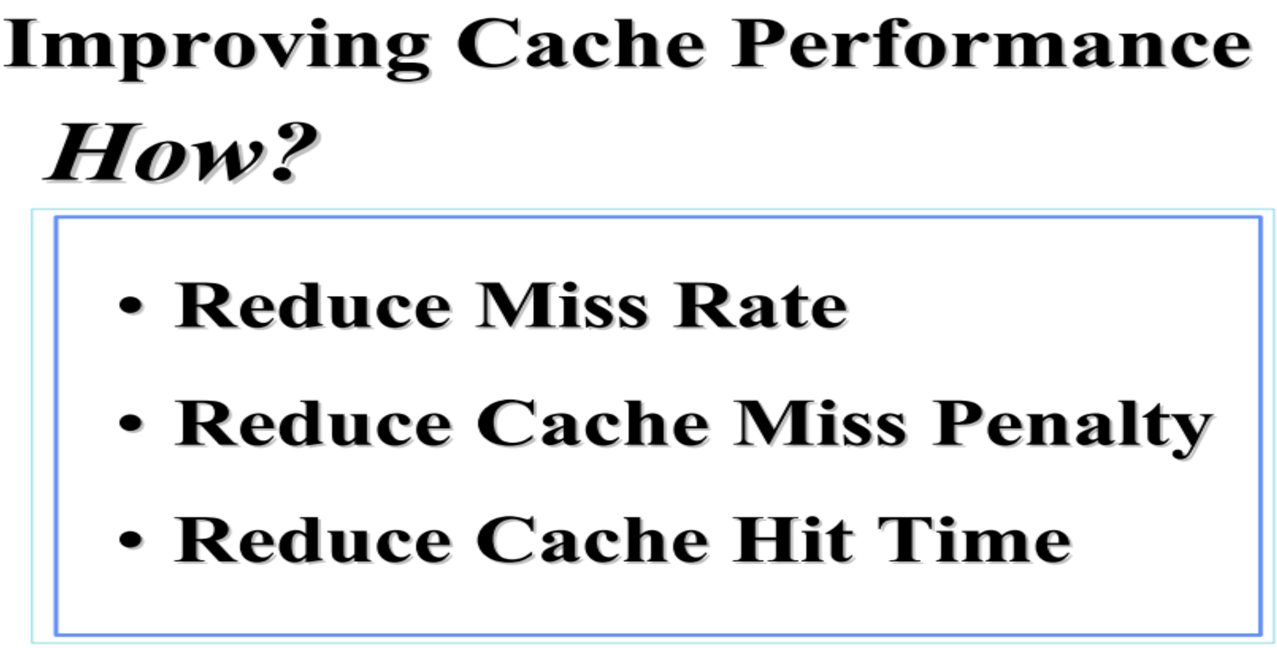
Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Example

❑ Set Associative Cache Organization



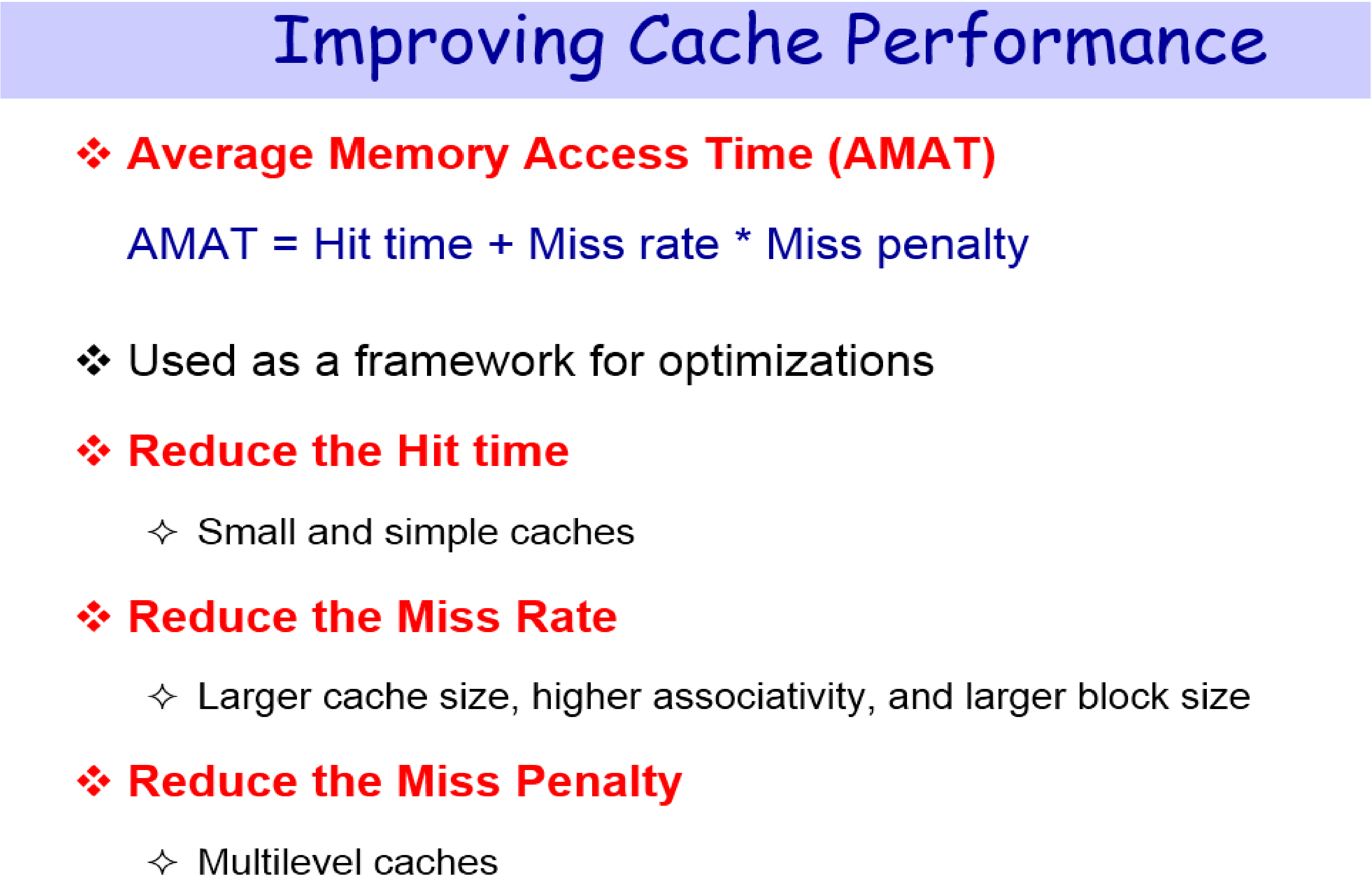
4-way set-associative cache with 4 comparators and one 4-to-1 multiplexor: size of cache is 1K blocks = 256 sets * 4-block set size



Improving Cache Performance

How?

- Reduce Miss Rate
- Reduce Cache Miss Penalty
- Reduce Cache Hit Time



Improving Cache Performance

❖ **Average Memory Access Time (AMAT)**

AMAT = Hit time + Miss rate * Miss penalty

❖ Used as a framework for optimizations

❖ **Reduce the Hit time**

✧ Small and simple caches

❖ **Reduce the Miss Rate**

✧ Larger cache size, higher associativity, and larger block size

❖ **Reduce the Miss Penalty**

✧ Multilevel caches

Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
  - if miss occurs in primary cache second-level cache is accessed
  - if data is found in L-2 cache miss penalty is access time of L-2 cache which is much less than main memory access time
- Main memory services L-2 cache misses
  - if miss occurs again at L-2 then main memory access is required and large miss penalty is incurred
- Some high-end systems include L-3 cache

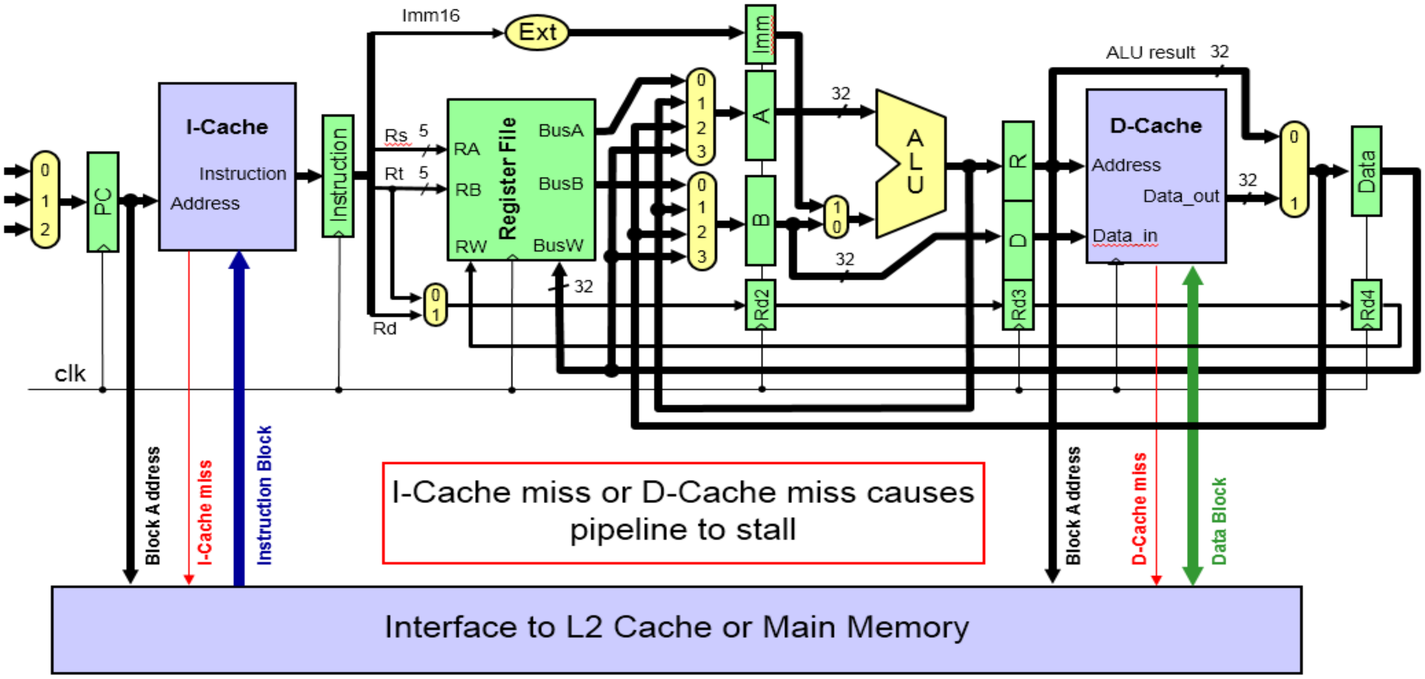
- **Example:** given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100 ns
  - With just primary cache:
    - Miss penalty = 100ns/0.25ns = 400 cycles
    - Effective CPI = 1 + 0.02 × 400 = 9
- Adding L-2 cache
  - Access time = 5 ns
  - Global miss rate to main memory = 0.5%
- miss penalty to L-2 cache (with L-2 hit)= 5ns / 0.25ns = 20 cycles
- Effective CPI = Base CPI + Primary stalls per instr. + Secondary stall per instr. = 1 + 2% x 20 + 0.5% x 400 = 3.4
- Performance ratio
  - machine with L-2 cache is faster by a factor of 9/3.4 = 2.6

Handling Cache Misses

- **cache miss** A request for data from the cache that cannot be filled because the data is not present in the cache.

- The control unit must detect a **miss** and process the miss by fetching the requested data from memory (or, as we shall see, a lower-level cache). Cache sends a **miss signal** to **stall** the processor.

- If the cache reports a **hit**, the computer continues using the data as if **nothing happened**. Consequently, we can use the same basic control that we developed in (The processor: datapah and control , pipelining). The memories in the datapath are simply replaced by caches.



- Modifying the control of a processor to handle a hit is trivial; misses, however, require some extra work.

- For a cache miss, we can **stall the entire processor**, essentially freezing the contents of the temporary and programmer-visible registers, while we wait for memory. In contrast, pipeline stalls, discussed in last chapter, are more complex because we must continue executing some instructions while we stall others.

We can now define the steps to be taken on an instruction cache miss:

1. Send the original PC value (current PC – 4) to the memory.

2. Instruct main memory to perform a read and wait for the memory to complete its access.

3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.

4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.

The control of the cache on a data access is essentially identical: on a miss, we simply stall the processor until the memory responds with the data.

Handling Writes

When CPU writes to cache, we may use one of two policies:

1- Write Through (Store through)

- Write through is a storage method in which data is written into the cache and the corresponding main memory location at the same time **(every write)**. The cached data allows for fast retrieval on demand, while the same data in main memory ensures that nothing will get lost if a crash, power failure, or other system disruption occurs.

- The other key aspect of writes is what occurs on a write miss. We first fetch the words of the block from memory. After the block is fetched and placed into the cache, we can overwrite the word that caused the miss into the cache block. We also write the word to main memory using the full address.

- Although write through minimizes the risk of data loss, every write operation must be done twice, and this redundancy takes time. (very simple but not provide very good performance).
  - **Solution**: performance is improved with a **write buffer**.
  - **Write buffer**: A queue that holds data while the data is waiting to be written to memory.
  - CPU continues immediately.
  - Only stalls on write if write buffer is already full.

- Write through is the preferred method of data storage in applications where data loss cannot be tolerated, such as banking and medical device control. In less critical applications, and especially when data volume is large, an alternative method called write back.

2- Write Back:

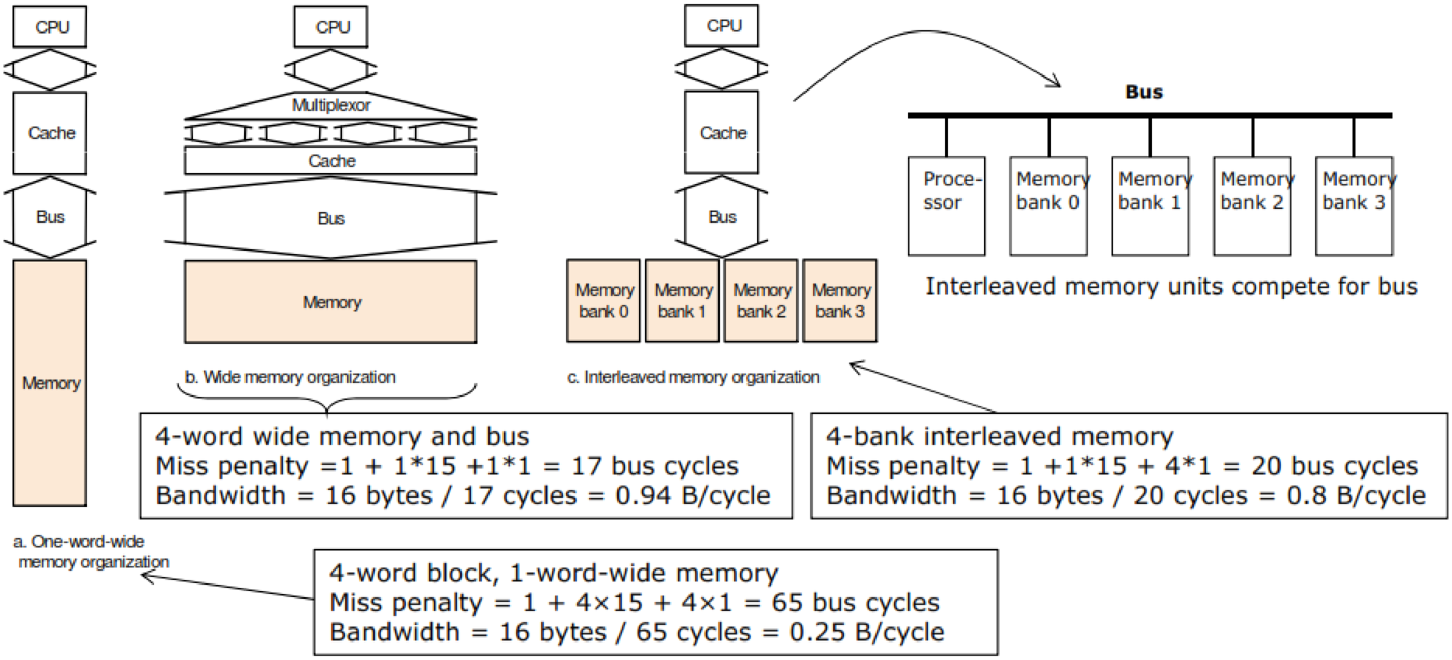
- Write back is a storage method in which data is written into the cache every time a change occurs, but is written into the corresponding location in main memory when it needs to be replaced or flushed.

- Write back optimizes the system speed because it takes less time to write data into cache alone, as compared with writing the same data into both cache and main memory (write through). **Write back** is more efficient than **write through**, but more complex to implement.

Improving Cache Performance

- **Increasing Memory Bandwidth:** Use DRAMs for main memory with Fixed width (e.g., 1 word).

  Example: Assuming cache block of 4 words
  - 1 clock cycle for address transfer (1 bus trip)
  - 15 clock cycles for each memory data access
  - 1 clock cycle per data transfer (1 bus trip)



Interleaved memory units compete for bus

a. One-word-wide memory organization

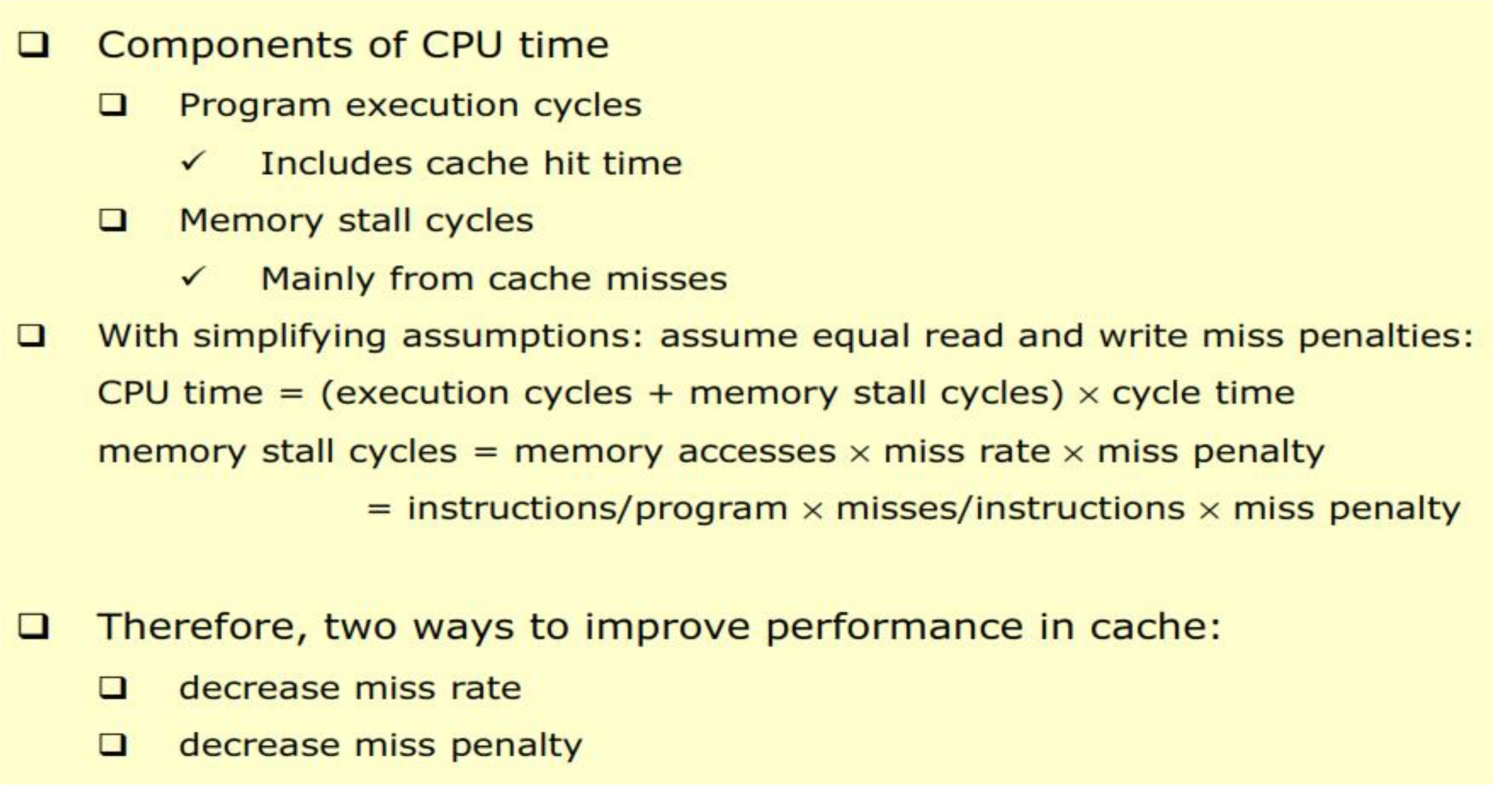
b. Wide memory organization

c. Interleaved memory organization

4-word wide memory and bus
Miss penalty =1 + 1*15 +1*1 = 17 bus cycles
Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle

4-bank interleaved memory
Miss penalty = 1 +1*15 + 4*1 = 20 bus cycles
Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

4-word block, 1-word-wide memory
Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles
Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle



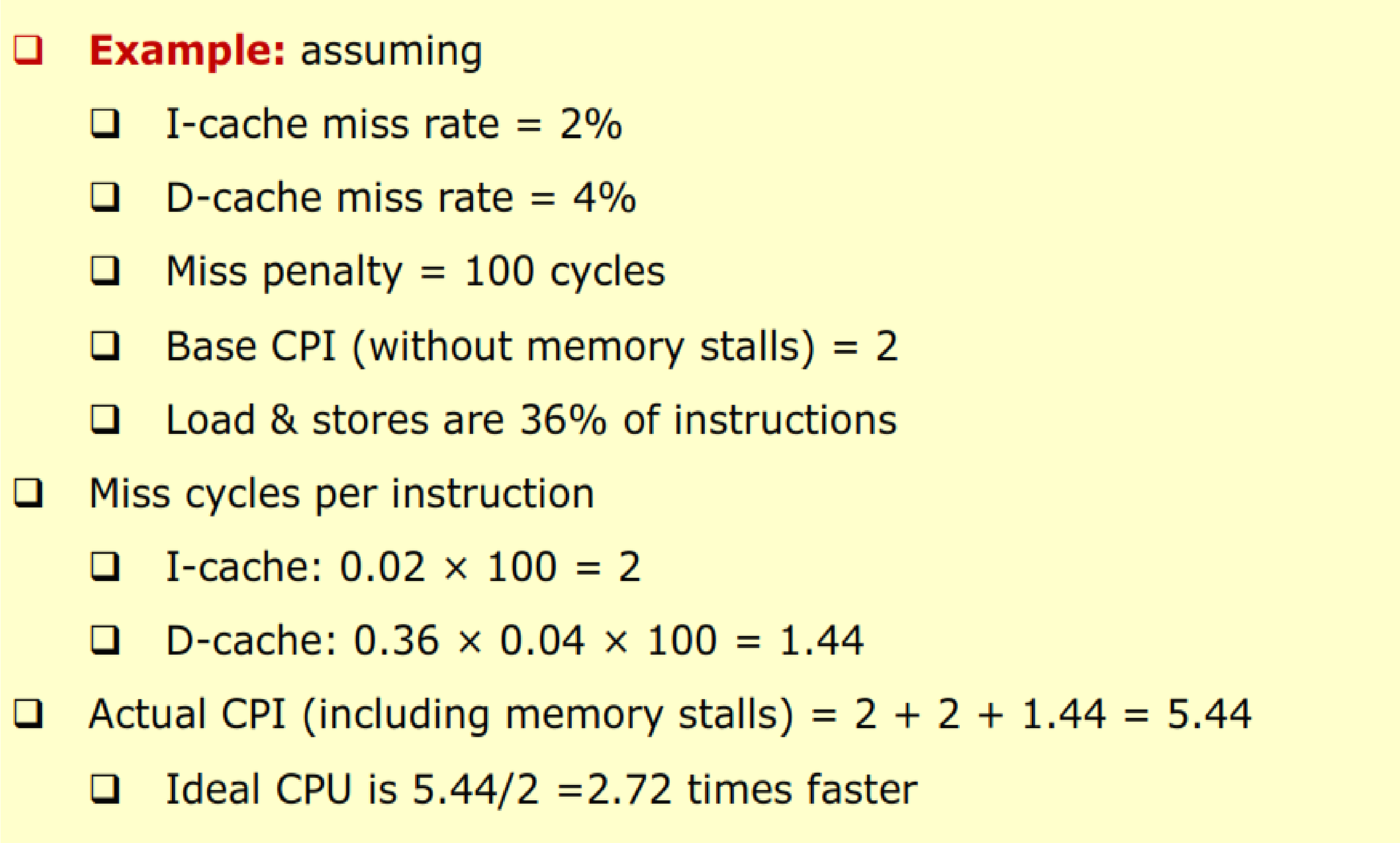
- □ Components of CPU time
  - □ Program execution cycles
    - ✓ Includes cache hit time
  - □ Memory stall cycles
    - ✓ Mainly from cache misses
- □ With simplifying assumptions: assume equal read and write miss penalties:

  CPU time = (execution cycles + memory stall cycles) × cycle time

  memory stall cycles = memory accesses × miss rate × miss penalty

  = instructions/program × misses/instructions × miss penalty

- □ Therefore, two ways to improve performance in cache:
  - □ decrease miss rate
  - □ decrease miss penalty



- □ **Example:** assuming
  - □ I-cache miss rate = 2%
  - □ D-cache miss rate = 4%
  - □ Miss penalty = 100 cycles
  - □ Base CPI (without memory stalls) = 2
  - □ Load & stores are 36% of instructions
- □ Miss cycles per instruction
  - □ I-cache: 0.02 × 100 = 2
  - □ D-cache: 0.36 × 0.04 × 100 = 1.44
- □ Actual CPI (including memory stalls) = 2 + 2 + 1.44 = 5.44
  - □ Ideal CPU is 5.44/2 =2.72 times faster